



OULUN YLIOPISTO  
UNIVERSITY of OULU

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

**Jarmo Luomala**

# **A TOOL FOR GENERATING PROTOCOL DISSECTORS FOR WIRESHARK IN LUA**

Master's Thesis  
Degree Programme in Computer Science and Engineering  
November 2013



OULUN YLIOPISTO  
UNIVERSITY of OULU

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING  
DEGREE PROGRAMME IN COMPUTER SCIENCE AND ENGINEERING

# **A TOOL FOR GENERATING PROTOCOL DISSECTORS FOR WIRESHARK IN LUA**

Author	_____
	Jarmo Luomala
Supervisor	_____
	prof. Juha Röning
Accepted	_____ / _____ 2013
Grade	_____

Luomala J. (2013) A tool for generating protocol dissectors for Wireshark in Lua. Department of Computer Science and Engineering, University of Oulu, Oulu, Finland. Master's thesis, 78 p.

## ABSTRACT

Packet analysis is an essential part of monitoring and understanding network traffic. Packet analyzers are used to capture binary data flowing inside networks, decode it, and parse the decoded stream into a structured, human-readable form. Packet analyzers consist of protocol dissectors, small pieces of software that dissect the captured data stream into separate packets and fields according to the specified protocol rules. Because new protocols are developed constantly and a dissector is needed for every protocol that an analyzer is supposed to understand, there is a continuous need to create dissectors. Manual writing of them is time-consuming, requires familiarity with the target analyzer, the structure of the dissectors, and of course programming skills. Therefore, a tool that would automate the actual dissector creation process would be useful.

In this thesis, packet analyzers and protocol dissectors are studied. Functionalities and typical uses of packet analyzers are explored, especially from the information security point of view. In the empirical part, a tool for generating protocol dissectors for Wireshark, which is a very popular network packet analyzer, is implemented and introduced. The tool has a graphical user interface and it generates the dissectors in Lua programming language. Several executed test cases with custom protocols have proven that the tool generates valid Lua files which work properly as protocol dissectors in Wireshark. Used sample packets are dissected correctly according to the protocol rules and definitions specified with the tool. Both fixed and variable length fields and packets can be defined, and both parent dissectors and sub-dissectors can be created.

Keywords: packet analyzer, protocol analyzer, packet sniffer, dissector

## TIIVISTELMÄ

Pakettianalyysi on olennainen osa tietoverkkoliikenteen tarkkailua ja ymmärtämistä. Pakettianalysointia käytetään kaappaamaan tietoverkoissa virtaavaa binääridataa, dekodamaan se ja parsimaan dekodattu data jäsenneltyyn, luettavaan muotoon. Pakettianalysointit koostuvat protokollapilkkoista/dissektoreista, pienistä ohjelmistopaloista, jotka leikkelevät kaapatun datan erillisiin paketteihin ja kentiin määriteltyjen protokollasääntöjen mukaisesti. Koska uusia protokollia kehitetään lakkaamatta ja analysointia tarvitsee dissektorin jokaista protokollaa varten, jota sen on tarkoitus ymmärtää, on jatkuva tarve luoda dissektoreita. Niiden kirjoittaminen käsin on aikaavievää, vaatii perehtyneisyyttä kohdeanalysointia, dissektoreiden rakentamiseen ja tietysti ohjelmointitaitoja. Sen vuoksi työkalu, joka automatisoi varsinaisen dissektorinluomisprosessin, olisi hyödyllinen.

Tässä diplomityössä tutkitaan pakettianalysointia ja protokolladissektoreita. Pakettianalysointia ja protokolladissektoreiden toiminnallisuutta ja tyypillisiä käyttötapoja tarkastellaan erityisesti tietoturvan näkökulmasta. Työssä kehitetään ja esitellään työkalu dissektoreiden tuottamiseksi Wireshark:lle, joka on erittäin suosittu tietoverkkopakettianalysointia. Työkalulla on graafinen käyttöliittymä ja se luo dissektoreita Lua-ohjelmointikielellä. Useat suoritettavat testitapaukset erikoisprotokollilla ovat osoittaneet, että työkalu luo valideja Lua-tiedostoja, jotka toimivat asianmukaisesti protokolladissektoreina Wireshark:ssa. Käytetyt näytepaketit pilkotaan oikein työkalulla määriteltyjen protokollasääntöjen ja määritelmien mukaisesti. Sekä kiinteän että vaihtelevan pituisia kenttiä ja paketteja voidaan määritellä, ja sekä yli- että alidissektoreita voidaan luoda.

Avainsanat: pakettianalysointia, protokolla-analysointia, pakettinuuksija, protokollapilkkoja, dissektoria

# TABLE OF CONTENTS

ABSTRACT

TIIVISTELMÄ

TABLE OF CONTENTS

FOREWORD

ABBREVIATIONS

<b>1. INTRODUCTION</b>	<b>9</b>
1.1. Research question . . . . .	10
<b>2. COMPUTER NETWORKS</b>	<b>11</b>
2.1. The OSI reference model . . . . .	11
2.2. The TCP/IP reference model . . . . .	11
2.3. Tanenbaum's hybrid reference model . . . . .	12
2.3.1. Physical layer . . . . .	13
2.3.2. Data link layer . . . . .	14
2.3.3. Network layer . . . . .	14
2.3.4. Transport layer . . . . .	15
2.3.5. Application layer . . . . .	15
2.4. End-to-end communication between hosts . . . . .	16
2.5. Summary . . . . .	17
<b>3. PACKET ANALYZERS</b>	<b>18</b>
3.1. What is a packet analyzer? . . . . .	18
3.2. Typical uses . . . . .	19
3.3. Widely used packet analyzers . . . . .	20
3.3.1. Wireshark . . . . .	20
3.3.2. Cain & Abel . . . . .	21
3.3.3. tcpdump . . . . .	21
3.3.4. Ettercap . . . . .	22
3.3.5. Clarified Analyzer . . . . .	22
3.3.6. Summary . . . . .	22
<b>4. INFORMATION SECURITY ASPECTS</b>	<b>24</b>
4.1. Critical characteristics of information – The C.I.A. triangle . . . . .	24
4.1.1. Confidentiality . . . . .	24
4.1.2. Integrity . . . . .	24
4.1.3. Availability . . . . .	25
4.2. Compromising information security with packet sniffers . . . . .	25
4.2.1. Man-in-the-Middle attack . . . . .	26
4.2.2. ARP spoofing . . . . .	26
4.2.3. DNS spoofing . . . . .	27

4.2.4.	Session hijacking . . . . .	28
4.2.5.	Replay attack . . . . .	30
4.3.	Protecting information security with packet sniffers . . . . .	30
4.3.1.	Network troubleshooting . . . . .	30
4.3.2.	Packet filtering . . . . .	31
4.3.3.	Intrusion detection . . . . .	31
4.3.4.	Data leakage detection . . . . .	32
<b>5.</b>	<b>WIRESHARK DISSECTORS</b>	<b>33</b>
5.1.	Dissectors written in C . . . . .	33
5.2.	Dissectors written in Lua . . . . .	33
5.3.	Dissector generators . . . . .	36
5.3.1.	CSjark . . . . .	37
5.3.2.	Interpreted dissector . . . . .	37
<b>6.</b>	<b>LUA DISSECTOR GENERATOR</b>	<b>38</b>
6.1.	Functionality and logic . . . . .	38
6.2.	Implementation . . . . .	42
6.2.1.	Programming language for the tool . . . . .	42
6.2.2.	Programming language for generating the dissectors . . . . .	42
6.2.3.	Development environment and tools . . . . .	42
6.3.	User interface and usage . . . . .	43
6.3.1.	How to run the software? . . . . .	43
6.3.2.	Input files . . . . .	43
6.3.3.	Main window . . . . .	44
6.3.4.	Help window . . . . .	48
6.3.5.	Define protocol information window . . . . .	49
6.3.6.	Define delimiters window . . . . .	51
6.3.7.	Define field information windows . . . . .	51
6.3.8.	Create protocol dissector window . . . . .	54
6.3.9.	Popup windows . . . . .	56
6.3.10.	How to use the generated Lua dissectors in Wireshark? . . . . .	57
<b>7.</b>	<b>EVALUATION OF LUA DISSECTOR GENERATOR</b>	<b>59</b>
7.1.	Test case 1 . . . . .	59
7.2.	Test case 2 . . . . .	62
7.2.1.	Generating the parent dissector . . . . .	62
7.2.2.	Generating the subdissector . . . . .	65
7.3.	Discussion . . . . .	67
<b>8.</b>	<b>CONCLUSION</b>	<b>68</b>
<b>9.</b>	<b>REFERENCES</b>	<b>69</b>
<b>10.</b>	<b>APPENDICES</b>	<b>73</b>

## FOREWORD

The Oulu University Secure Programming Group (OUSPG) is an academic research group in the Department of Computer Science and Engineering at the University of Oulu. Since 1996, it has been studying, evaluating, and developing methods of testing software in order to discover and eliminate implementation level vulnerabilities and other security issues. One of the OUSPG's main research fields has been protocol implementations, a subject that also this thesis tackles.

I want to express my gratitude to my supervisor, prof. Juha Röning, for giving me the opportunity to work on my thesis at the OUSPG and for providing feedback. Special thanks deserve also my second supervisor, prof. Timo Ojala, for reviewing the thesis, and my mentor Christian Wieser, for coming up with the idea for the empirical part and for giving me valuable tips and guidance during the process, and Marko Laakso for helping me with the research question. Thanks for the last tips and kicks on the homestretch go to Thomas Wahlberg. And to the whole fellowship of the OUSPG: Thank you for the great atmosphere to work in, your help and support, and all the fun times!

Finally, I would like to thank my parents and siblings for their support during my studies, and my wife, Kati, for being always loving and supportive and pushing me forward in the upward slopes. Kiitos.

Oulu, Finland November 24, 2013

Jarmo Luomala

## ABBREVIATIONS

APDU	Application PDU
APR	ARP Poison Routing
ARP	Address Resolution Protocol
ARPANET	Advanced Research Projects Agency Network
ASCII	American Standard Code for Information Interchange
BNFC	Backus-Naur Form Converter
CWE	Common Weakness Enumeration
DDoS	Distributed Denial-of-Service
DHCP	Dynamic Host Configuration Protocol
DLP	Data leakage/loss prevention
DNS	Domain Name System
DoS	Denial-of-Service
FBI	Federal Bureau of Investigation
GNU	GNU's Not Unix
GPL	GNU General Public License
GUI	Graphical user interface
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
ICMP	Internet Control Message Protocol
ID	Identifier
IDS	Intrusion detection system
IEEE	Institute of Electrical and Electronics Engineers
IP	Internet Protocol
IPS	Intrusion prevention system
IRC	Internet Relay Chat
ISO	International Standards Organization
LAN	Local Area Network
Lex	Lexical analyzer
MAC	Media Access Control
MITM	Man-in-the-Middle
NIC	Network Interface Card/Controller
OS	Operating system
OSI	Open Systems Interconnection
OUSPG	Oulu University Secure Programming Group
P2P	Peer-to-Peer
PDU	Protocol Data Unit
PLY	Python Lex-Yacc
PyYAML	Python YAML
QoS	Quality of Service
RR	Resource record
SANS	SysAdmin, Audit, Networking, and Security Institute
SQL	Structured Query Language
SSH	Secure Shell
SSL	Secure Sockets Layer
TCP	Transmission Control Protocol



TLS	Transport Layer Security
UDP	User Datagram Protocol
URL	Uniform Resource Locator
UTC	Coordinated Universal Time
WAN	Wide Area Network
WWW	World Wide Web
XSS	Cross-site Scripting
Yacc	Yet Another Compiler-Compiler
YAML	YAML Ain't Markup Language

# 1. INTRODUCTION

*“All network problems stem from the packet level, where even the prettiest-looking applications can reveal their horrible implementations and seemingly trustworthy protocols can prove malicious. To better understand and solve network problems, we go to the packet level where nothing is hidden from us, where nothing is obscured by misleading menu structures, eye-catching graphics, or untrustworthy employees. Here there are no secrets, and the more we can do at the packet level, the more we can control our network and solve problems. This is the world of packet analysis.”* [1, p. 1]

— Chris Sanders, a security researcher and author

Computers and networks are essential parts of our daily lives. Networks are based on protocols which define the format of the messages and rules for exchanging them between systems. It has been said that protocols are to communications what programming languages are to computations: protocols define the syntax and semantics of communications.

Network analysis (also known as protocol analysis, packet analysis, or simply sniffing) is the process of capturing network traffic and analyzing it in detail to determine what is happening on the network. A packet analyzer decodes, or dissects, the captured data packets that follow protocol definitions known to the analyzer, and displays the network traffic in a human-readable format [2].

Packet analyzers are efficient tools in the right hands. Undeniably many of them are actively used by malicious hackers to sniff sensitive information and help them break into computer systems, but they were originally designed for testing and monitoring networks, devices attached to them, and software implementations running on them [1, 3, 4]. Packet analyzers are very useful in detecting design flaws and implementation bugs in network software before possible vulnerabilities are exploited and privacy is compromised. Currently, the leading packet analyzer in the world and the de facto standard across many industries and academic institutions is Wireshark [5]. For that reason, Wireshark was chosen as the subject of this thesis and the target analyzer for which the dissectors are to be generated.

Dissectors are small pieces of code that are utilized by a packet analyzer to dissect and analyze captured network data stream. A packet analyzer needs a dissector for every protocol the packets of which it should be able to understand. There are already a huge number of protocols out there, but yet new ones are constantly developed and some of them are even proprietary. So there is and will be a need to create dissectors for new protocols, in order to monitor networks and detect malicious packets, reveal poor implementations, and solve other occurring problems.

The purpose of this thesis is to study the theory and different uses of packet analyzers, especially from the information security point of view, and implement a tool with a graphical user interface for creating custom protocol dissectors for Wireshark. Manual writing of dissectors can be a time-consuming and tedious task. Thus it would be great to have a tool that would make the creating of

dissectors for custom protocols faster without the need to know the specifics of how to actually code a dissector or even without programming skills at all. The user only needs to know, or guess, the protocol specifics, in other words, how a packet of the protocol is constructed.

In Figure 1, the number of IEEE (Institute of Electrical and Electronics Engineers) articles that are published in conferences, journals, and magazines, and that study network security or information security is presented [6]. The number of published articles is divided into time frames of three years. It can be noticed clearly that the research in the field of information and network security is constantly growing. That is partly why this thesis also examines packet analyzers and dissectors from the information security point of view.

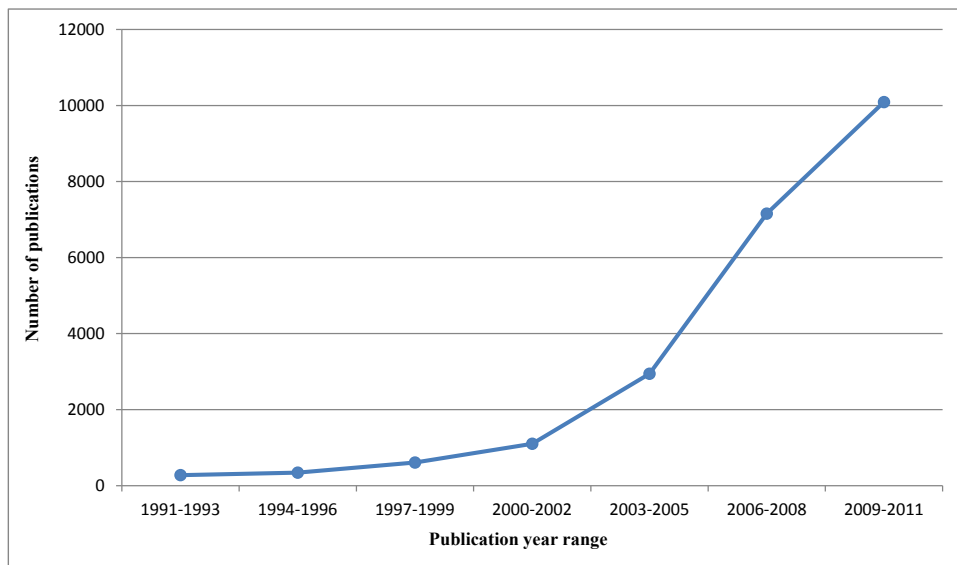


Figure 1. Articles published by IEEE that contain keywords “*network security*” OR “*information security*” in their metadata.

### 1.1. Research question

This thesis and the developed software answer the question “Is it possible to create a tool that would automate protocol dissector generation for Wireshark without requiring programming skills from the user?”. The question is answered by introducing the implemented tool, explaining its usage, verifying its functionality with a couple of test cases, and utilizing the generated dissectors in Wireshark.

## 2. COMPUTER NETWORKS

Most computer networks are organized as layers that are built on top of each other. The purpose of this stack of layers is to reduce the complexity of designing networks and implementing network software. Each layer  $N$  implements some specific services and provides them for the higher level layer  $N + 1$ . A layer hides the implementation details of its services and makes them available through well-defined interfaces. Every layer contains a set of alternative protocols, but only one protocol per layer is used in an end-to-end connection. [7, Ch. 1.3]

A set of layers and protocols is called a network architecture. In the following sections, three different network architectures are introduced. Only Tanenbaum's hybrid model and all its layers are described in detail and the other two are merely mentioned for comparison and their publicity.

### 2.1. The OSI reference model

The OSI (Open Systems Interconnection) reference model was developed by the International Standards Organization (ISO). The name "Open Systems Interconnection" refers to the fact that the model defines the principles of connecting systems that are open for communication. The OSI model aims to make a distinction between three central concepts of network architecture: services, interfaces, and protocols. The services define the duties of each layer, the interfaces define how to access these services, and the protocols are the ones that perform the actual services inside the layer. The model merely specifies the functions that each layer should fulfill, not the exact protocols to be used. It consists of seven layers, as shown in Table 1. [7, Ch. 1.4]

Table 1. Layers of the OSI reference model

7	Application layer
6	Presentation layer
5	Session layer
4	Transport layer
3	Network layer
2	Data link layer
1	Physical layer

### 2.2. The TCP/IP reference model

The TCP/IP [8] (Transmission Control Protocol / Internet Protocol) reference model (also known as the Internet protocol suite<sup>1</sup>) was originally designed for

---

<sup>1</sup>[http://en.wikipedia.org/wiki/Internet\\_protocol\\_suite](http://en.wikipedia.org/wiki/Internet_protocol_suite)

the ancestor of the Internet and all wide area networks (WANs), the ARPANET (Advanced Research Projects Agency Network). The name of the model comes from its two primary protocols: the Transmission Control Protocol and the Internet Protocol. The name also implies the fact that the protocols were invented before the model, thus making it unsuitable for any other protocol stacks. Compared with the OSI model, the TCP/IP model fails to distinguish clearly the three concepts mentioned in Section 2.1. The model is visualized in Table 2. [7, Ch. 1.4]

Table 2. Layers of the TCP/IP reference model

4	Application layer
3	Transport layer
2	Internet layer
1	Host-to-network layer

### 2.3. Tanenbaum's hybrid reference model

Tanenbaum's model is constructed, as the name implies, by combining the OSI reference model and the TCP/IP reference model. See Table 3 for the correspondence of the layers of the two models.

Table 3. Comparison of the OSI model and the TCP/IP model

	<i>OSI</i>	<i>TCP/IP</i>
7	Application layer	Application layer
6	Presentation layer	—
5	Session layer	—
4	Transport layer	Transport layer
3	Network layer	Internet layer
2	Data link layer	Host-to-network layer
1	Physical layer	

The reason why this hybrid model is introduced in this thesis is that the other two have had some problems and criticism directed at them. In short, the layers of the OSI model (except layers 5 and 6) have proven to be useful, but the protocols have not become popular. On the other hand, the case of the TCP/IP model is quite the opposite: the model is nearly obsolete, but the protocols are still widely in use. [7, Ch. 1.4]

Tanenbaum's hybrid model is illustrated in Figure 2. In theory, layer N on host A communicates with layer N on host B, but, in practice, no data transfers directly between layers of the same level on different machines. Instead, each layer adds its own header information to every payload and passes them on to

the next layer below. When layer 1 finally receives the payload, it injects the data to the actual physical communication link (wired or wireless), which delivers the data to the receiver. At the receiving end, the process goes inversely and the payload is decapsulated step by step. In the figure, physical communication is shown by solid lines and virtual communication by dotted lines. [7, Ch. 1.3]

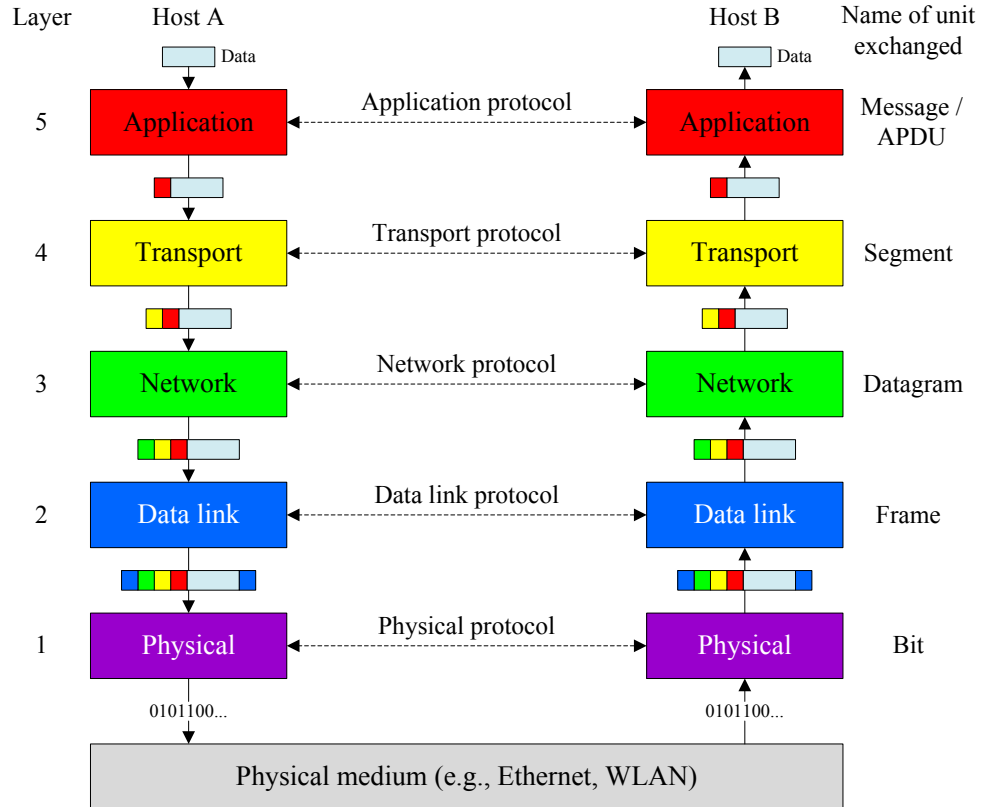


Figure 2. Tanenbaum's hybrid reference model.

### 2.3.1. Physical layer

All networks and the subsequent layers are based on the physical layer. It is in charge of transmitting raw binary data, *bits*, over a communication channel. It handles [7, Ch. 2]

- (a) the conversion of the digital data (bits) into corresponding electrical signals, and vice versa,
- (b) the establishment and teardown of a connection,
- (c) timing related issues and other physical specifics.

### 2.3.2. Data link layer

The data link layer is concerned with getting *frames* from one end of a communication link to the other. The layer has many tasks, but the most important ones are [7, Ch. 3]

- (a) *Framing* – The data link layer splits the bit stream it receives from the physical layer into a sequence of frames. The purpose of this process is to enable the detection and correction of possible transmission errors, for example, by providing a computed checksum for each sent frame or by adding some redundant bits. The actual framing is usually done with flags and bit/byte stuffing.
- (b) *Error control* – In the case of a reliable, connection-oriented service, such as TCP, all the frames sent by host A have to be delivered successfully in the proper order to the network layer of host B. Acknowledgment of successfully received frames and retransmission of damaged or lost frames is done in the data link layer.
- (c) *Flow control* – A mechanism to prevent the sender from overloading the receiver with incoming frames. The mechanism can be based on the feedback from the receiver or be built in the used protocol (sliding window protocols). Flow control is performed also in the upper layers.

The data link layer contains typically a sublayer called the Media Access Control (MAC) sublayer. This layer is particularly important in IEEE 802 standard LANs (local area network). It makes it possible for many hosts to use a shared communication medium simultaneously. The MAC sublayer divides the channel into several subchannels, which are allocated dynamically as needed, and handles occurring collisions. [7, Ch. 4]

### 2.3.3. Network layer

The network layer is responsible for delivering *datagrams* all the way from the source to the destination. The process of transferring packets through intermediate network nodes and possibly through several separate networks before reaching the destination is called *routing*. Many different networks exist with a variety of protocols. The art of connecting these networks by routers is called internetworking. This layer provides connectionless, best-effort service to the transport layer. [7, Ch. 5]

The Internet uses many protocols related to the network layer, but obviously the most important of them is the Internet Protocol (IP). It is the foundation of the World Wide Web (WWW) and many other networking applications, for example, email, instant messaging, multimedia services, and P2P (peer-to-peer) applications. In addition, the network layer includes also routing protocols and several control protocols, such as ICMP (Internet Control Message Protocol), ARP (Address Resolution Protocol), and DHCP (Dynamic Host Configuration Protocol). [7, Ch. 5]

### 2.3.4. *Transport layer*

The main purpose of the transport layer is to establish a logical end-to-end connection between application processes, in which data is exchanged in the form of *segments*. The transport layer is very similar to the network layer and many of their services are basically the same (e.g., addressing, flow and error control). The main difference is that the transport layer software runs on the end users' machines, and the network layer software runs mostly on the routers and other network devices. The objective is to enhance the best-effort service of the network layer, ensure the quality of service, and provide the possibility to have reliable connections on top of unreliable, error-prone networks. [7, Ch. 6]

The Internet has two primary transport protocols: UDP (User Datagram Protocol) and TCP (Transmission Control Protocol). The former is an unreliable, connectionless protocol, but the latter provides a reliable, connection-oriented service. UDP is suitable for some client-server interactions and real-time multimedia applications, but TCP is used by applications that demand reliability, such as file transfer and email. Both TCP and UDP communications are carried out through end points called sockets. Hosts use IP addresses and port numbers to direct segments to appropriate sockets. Port numbers below 1024 (well-known ports) are reserved for standard services. [7, Ch. 6]

### 2.3.5. *Application layer*

The application layer interacts directly with the network applications that implement the functions executed by the user. An application on host A communicates with another application on host B by sending *messages* to each other with the help of the underlying layers that take care of the actual transfer of the encapsulated messages. Application layer protocols define the syntax and semantics of the messages exchanged between applications. [7, Ch. 7]

The Internet offers numerous useful applications, but arguably two of the most essential ones are email (electronic mail) and the World Wide Web. The World Wide Web (often confused with the Internet) is a structured system of interlinked hypertext documents, which can contain many kinds of multimedia content, such as images, video and audio files. These resources are often accessed via URLs (Uniform Resource Locators), the main parts of which are domain names. Domain names are character strings that are used by humans to address specific network resources, instead of numerical addresses that are hard to remember. The use of domain names is possible due to one of the most important services provided by the application layer, the Domain Name System (DNS), which handles the resolving of domain names to IP addresses. [7, Ch. 7]

In addition to WWW and email, there are also many other important and popular network applications. Instant messaging and voice over IP (VoIP) have made communications cheaper, P2P applications have made file sharing easier, and.. [7, Ch. 1]



## 2.4. End-to-end communication between hosts

In computer networks when host A wants to send a packet to host B, it first checks its ARP (Address Resolution Protocol) cache for an appropriate IP-to-MAC address mapping. If host A does not know the MAC (Media Access Control) address of host B, it sends an ARP request to every host (broadcast) in its local area network (LAN). The ARP request contains the publicly known IP address of host B and it is requesting the corresponding MAC address of host B. The ARP request process is illustrated in Figure 3. [7, 9, 10, 1]

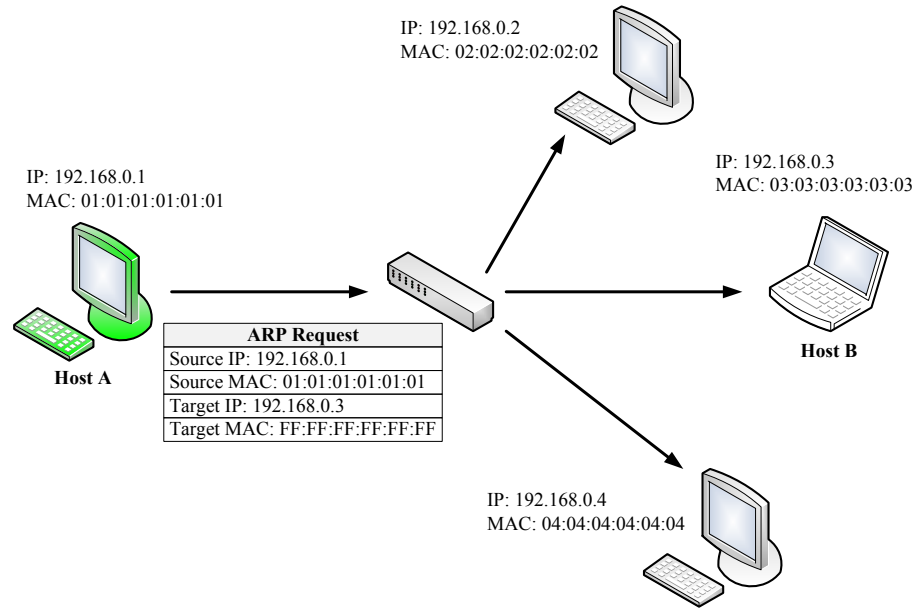


Figure 3. ARP request example.

The host who has the specified IP address, in this case host B, responds to host A with an ARP reply containing its MAC address, as shown in Figure 4. Hereafter host A and B communicate by using each other's MAC addresses. Each sent packet is still broadcasted to the entire LAN, but the network interface card (NIC) of a host checks the destination MAC address of the packet before accepting it. If the destination address matches the MAC address of the host's NIC, the packet is accepted, otherwise, it is discarded. However, packet sniffers have the ability to set the network interface card in promiscuous mode, in which it accepts all the packets flowing through the network segment, including those that are not addressed to the host running the sniffer. Hence, anyone can read the packets sent in the LAN, even those destined to the router. [9, 10, 1]

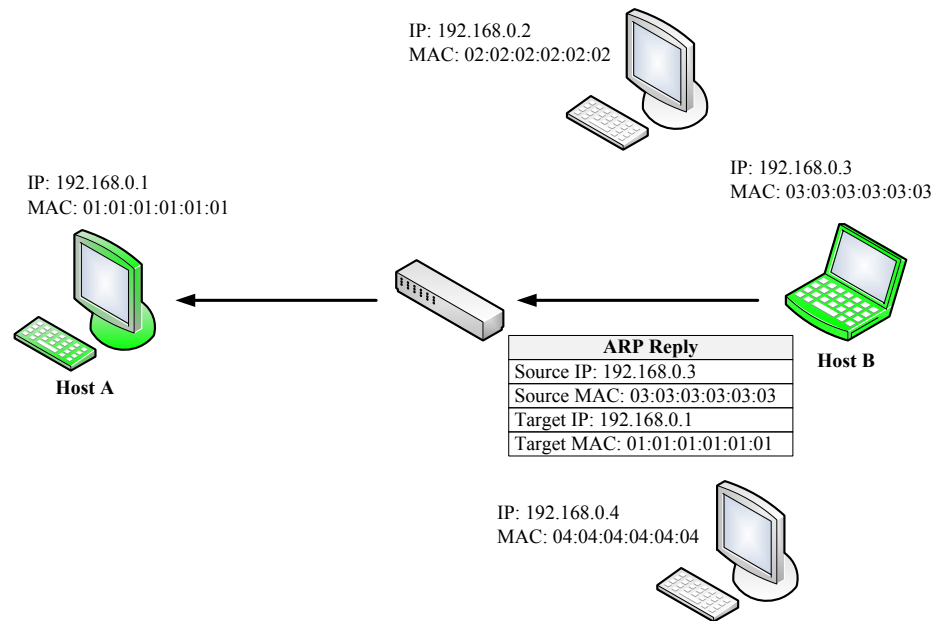


Figure 4. ARP reply example.

## 2.5. Summary

The previously described layers define the duties of protocols on different levels of a network architecture. Understanding the layered nature of networks and how information is transferred in networks is essential in order to realize the purpose and power of packet analyzers. Analyzers use dissectors as weapons to slice the network data stream into separate protocol data units (PDUs) and further into headers and payloads. During the analysis, most packet analyzers build an organized dissection tree according to the protocols of different layers. Wireshark, for example, builds the dissection tree upside down, so that the lowest layer protocol is shown at the top of the tree and the highest layer protocol at the bottom.

### 3. PACKET ANALYZERS

This chapter presents a brief introduction to packet analyzers. Section 3.1 explains what a packet analyzer is and how it is used in a network. Typical uses of packet analyzers are proposed and categorized in Section 3.2. Finally, some popular, widely used sniffers and analyzers are introduced in Section 3.3.

#### 3.1. What is a packet analyzer?

A packet analyzer, also known as a packet sniffer, a network analyzer, or a protocol analyzer, is a computer program used to tap a network and read binary data flowing inside it. Raw binary data, as such, is not human-readable, therefore a protocol analysis must be performed and the data must be dissected into separate packets and fields. Packet headers and contents are then decoded and interpreted according to appropriate encoding rules and specifications. This process is frequently referred to as (packet) sniffing, which tends to be one of the most popular terms in use nowadays [2]. [1]

The packet sniffing process can be divided into three consecutive steps: [1, Ch. 1]

1. *Capturing* – In the first step, the packet sniffer sets the network interface card (NIC) of the computer into *promiscuous mode*, in which the NIC can intercept all the traffic flowing through its network segment. With the help of this mode and low-level access to the network interface, the sniffer is able to capture and read the raw binary data from the wire (sniffing wireless connections is also possible though completely different).
2. *Decoding* – In the second step, the captured binary data is converted/decoded into a readable form. After this step, the network data may already be much more understandable to the user, even though it is hardly analyzed or interpreted at all.
3. *Analyzing* – The last, but definitely not the least, step involves the actual analysis of the captured and decoded data. In this step, the used protocol is interpreted from the information extracted from the network traffic. After the protocol is verified, the data is analyzed according to the protocol specification, resulting in a structured presentation of separate packets and their fields.

In Figure 5, an example of packet analysis (or protocol analysis) is performed for a simple HTTP (Hypertext Transfer Protocol) response message. The raw network data is shown in hexadecimal form instead of binary for the sake of compactness. A packet analyzer first decodes the sniffed data into a readable format and interprets the protocol of the packet. Then it dissects the packet into valid fields (headers and the payload/message body) and organizes them in a tree structure.

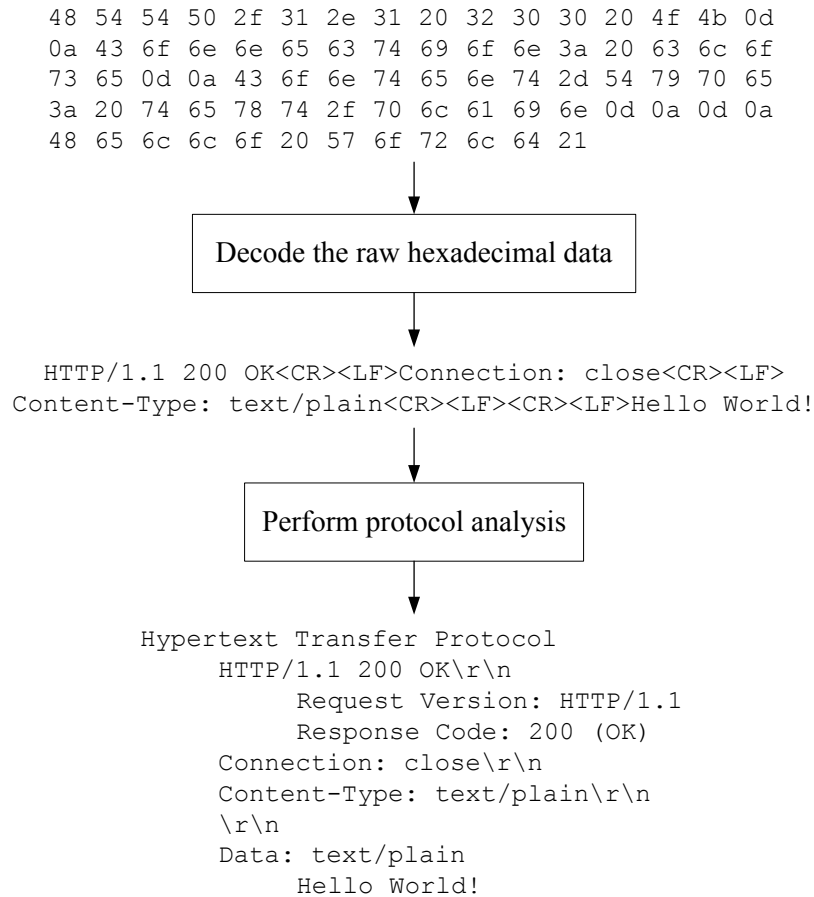


Figure 5. Protocol analysis performed for an HTTP response message.

### 3.2. Typical uses

Typical uses of packet sniffers and network analyzers can roughly be divided into two categories:

1. *Benevolent uses* – Uses that aim for protecting, testing, and administering the network.
2. *Malicious uses* – Uses that aim for eavesdropping for sensitive information, or even attacking and exploiting the network and its users.

Typical people who perform tasks that are categorized as benevolent can be, for example, network administrators, software testers, and information security personnel. For a network administrator, a packet analyzer is an efficient tool in troubleshooting network problems, monitoring network usage, and gathering statistics and traffic logs. Packet analyzers are also very useful in testing and verifying network software functionality, protocol implementations, and internal control system effectiveness (e.g., firewalls, filters, access controls). Uses that concentrate particularly on protecting information security include detecting network misuse and intrusion attempts, and filtering suspicious packets from network traffic. [9, 10, 11]

In addition to benevolent uses, packet analyzers and sniffers are utilized by hackers, especially black hat hackers, and crackers to conduct malicious actions. With the help of a sniffer it is rather easy to snoop on other network users' private conversations and messages. Usually, a hacker's objective is to obtain some sensitive information of the targeted network user or computer. Examples of such information are usernames, passwords, credit card numbers, cookies, and session IDs. Encryption is a necessary measure in protecting this kind of information, but still some websites send and receive data in plain text, which makes it easy to eavesdrop by simply sniffing the network traffic. Even if cryptographic protocols are used it is still possible to compromise security, for example, with Man-in-the-Middle (MITM) attack and reveal passwords by spoofing or pure brute-force cracking. [9, 12, 13, 11]

### 3.3. Widely used packet analyzers

There are many packet analyzers available for different purposes and equipped with different functionalities. Here some of the most prominent ones are introduced based on the listings provided on websites [14] and [15]. The last one, Clarified Analyzer, is mentioned due to its connection to the OUSPG: it is developed by Clarified Networks Oy, a spin-off company of the research group.

#### 3.3.1. *Wireshark*

Wireshark (formerly known as Ethereal) is the leading network protocol analyzer in the world. It is free, open source, and supports all common platforms. Wireshark provides many useful features in addition to the obvious: capturing live packet data from a network interface. Examples of those features are displaying packets with very detailed protocol information, packet filtering based on many criteria, display filters for over thousand protocols, and support for many different capture file formats. Captured network data can be browsed in its graphical user interface (GUI), see Figure 6, which has coloring rules for different packets and fields, thus making the packet analysis easier. There is also a command-line version of the tool, named TShark, which can be more suitable in certain situations, for example, when the packet capturing is to be automated and executed in a script. [5]

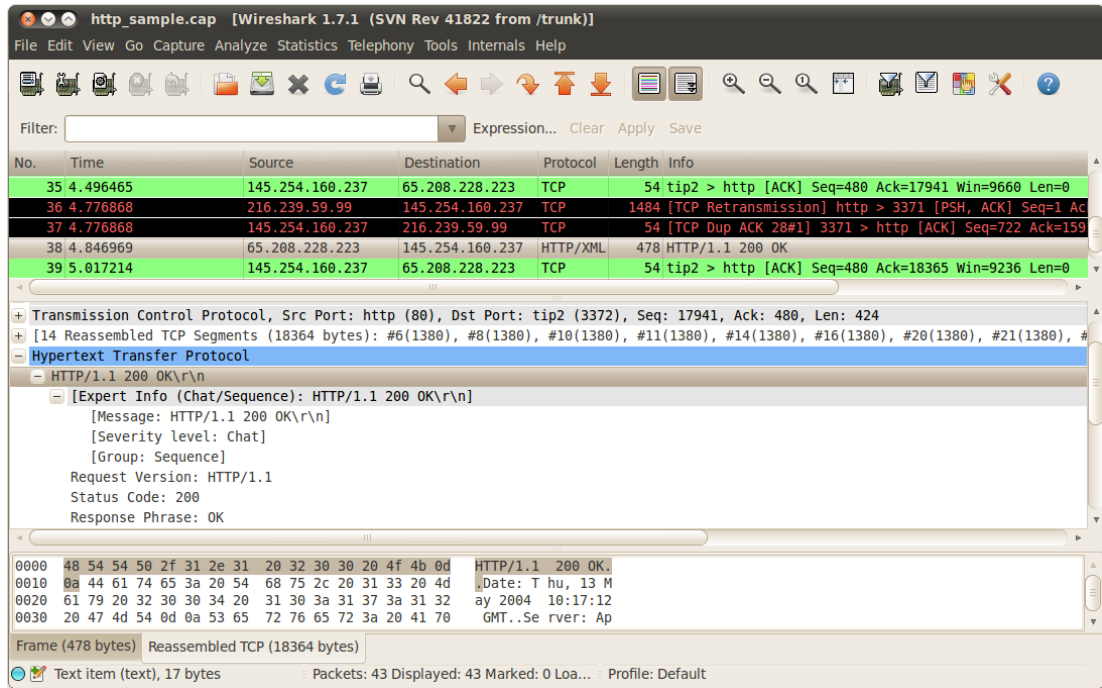


Figure 6. Screenshot of Wireshark’s graphical user interface.

### 3.3.2. *Cain & Abel*

Cain & Abel is a free password recovery tool for Microsoft Windows platforms. It is able to recover and reveal various kinds of passwords by simply sniffing the network traffic, or even cracking encrypted passwords and private keys using dictionary, brute-force, and cryptanalysis attacks. The latest version of the software contains APR (ARP Poison Routing) feature, which enables sniffing on switched networks and hijacking of IP traffic between hosts. Nevertheless, the tool is not meant to help exploiting software vulnerabilities or any other illegal activities, but to uncover weaknesses in protocol implementations, authentication methods, and caching mechanisms. [16]

### 3.3.3. *tcpdump*

tcpdump is a command-line based, multi-platform packet analyzer, which analyzes network behavior and monitors applications that process network traffic. tcpdump can be run with many different option flags, but its main function is to print out a description of the contents of the network packets that match a given Boolean expression. It is one of the oldest tools in the field and is also distributed as freeware. [17]

### 3.3.4. *Ettercap*

Ettercap is a comprehensive suite for Man-in-the-Middle attacks on LANs due to its ARP spoofing capability. It is free, open source, and cross-platform. Three different interfaces are offered to the user: a traditional command-line interface, a GUI, and ncurses<sup>1</sup>. It has many features, the most important ones being sniffing of live connections, performing network and host analysis, and content filtering on the fly, but the features can be further extended by adding new plugins. Ettercap supports both active and passive dissection of several protocols, including ciphered ones (e.g., HTTPS and SSH). [18]

### 3.3.5. *Clarified Analyzer*

Clarified Analyzer [19] is a commercial network analysis tool for network troubleshooting, traffic auditing, and abuse monitoring. The system consists of two parts: the Recorder and the Analyzer. The Recorder captures packets from taps, possibly filters them according to specified rules, and stores the complete packet history with computed flow indices on the disk. The captured packet data is simultaneously recorded and exported to the Analyzer, which then illustrates causal relationships between events with different visualizations (e.g., association graph, earth view, and timeline). The visualizations provide a better overall understanding of the network traffic and allow drilling down from a high-level overview down to the underlying details of the captured packet data. One of the main benefits of the Clarified Analyzer is the timeline functionality, which makes it possible to return to a specific incident for closer inspection when network problems occur. [20]

### 3.3.6. *Summary*

Clearly there are many kinds of packet analyzers with various features, and only some examples of them are introduced in this chapter. Other sniffers and network security tools worth mentioning are dsniff (old, no updates since 2000), Kismet, NetStumbler, Capsa, Carnivore (obsolete), Nmap (Network Mapper), and Metasploit.

Kismet [21] and NetStumbler [22] are targeted especially for sniffing traffic in wireless networks. Capsa [23], on the other hand, is a real-time network analyzer for both wired and wireless networks. Nmap [24] is a scanner for network mapping and security auditing, and dsniff [25] is a collection of network auditing and penetration testing tools. Metasploit [26] is a penetration testing software (including the framework and different editions) that helps in identifying security issues, managing vulnerabilities, and verifying mitigations. Carnivore (also known as DCS-1000) was a packet sniffing system developed by the Federal Bureau of Investigation (FBI) for the needs of law enforcement [2]. It was used to

---

<sup>1</sup><http://www.gnu.org/software/ncurses/>

monitor the network traffic (especially emails) of a suspect for evidence of illegal activities. The tool is nowadays obsolete and replaced with commercial software.

Common features to all packet analyzers are naturally the ability to capture and filter network packets in real-time, and examine individual packet contents at some level. Most sniffers available are even equipped with more than one user interface: a GUI for user-friendliness and expressiveness, and a command-line interface for efficiency and interoperability.

As mentioned earlier, there are both free and commercial tools available. When choosing a sniffer, one should consider and compare at least the reliability (Is it accurate? Does it miss packets?) and the usability (Is it easy to use?) of the tool. The cost is also an important factor in more than one way: the financial cost may obviously be relevant, but maybe even more relevant is the cost in the sense of interference. Meaning that does the analyzer interfere with the network traffic in any way? There are also big differences between sniffers in the number of supported protocols and operating systems. If custom protocols are used, one important feature to check is whether or not the analyzer is extendable with self-made dissectors. It was noted that this possibility is surprisingly rare in packet analyzers.

Some sniffers are designed mainly for passive network monitoring and troubleshooting, but some are targeted for active penetration testing and conducting attacks in order to discover vulnerabilities. Ettercap and Cain & Abel, for example, have ARP spoofing capability for performing MITM attacks, thus being able to reveal critical problems in the network security.



## 4. INFORMATION SECURITY ASPECTS

In this chapter, the information security aspects of packet analyzers are discussed. In Section 4.1, the critical characteristics of information and common attacks threatening them are introduced. Section 4.2 lists and describes some attacks in which packet sniffers may be utilized, and respectively, Section 4.3 lists and describes some situations where packet sniffers can help to protect security by facilitating the detection of malicious activities in networks and by helping administrators in network troubleshooting.

### 4.1. Critical characteristics of information – The C.I.A. triangle

The United States Code (U.S.C.), published by the Office of the Law Revision Counsel of the U.S. House of Representatives, defines information security as protecting and ensuring the three key concepts of information and information systems. These concepts are confidentiality, integrity, and availability. Together they form the so called C.I.A. triangle. [27]

In the following, these critical characteristics of information are explained briefly and some common attacks against each of them are mentioned.

#### 4.1.1. Confidentiality

According to the U.S.C., confidentiality “*means preserving authorized restrictions on access and disclosure, including means for protecting personal privacy and proprietary information*” [27]. In other words, confidentiality ensures that only the users with granted privileges are able to access the information. Confidentiality is closely related to privacy, and the value of privacy is especially high when it concerns personal information, credit card numbers, medical records, and other sensitive data. [28]

Violations of confidentiality may be unintentional disclosures or malicious attacks to computer systems and databases. One technique to steal sensitive information without getting caught is called “salami theft”, in which the attacker steals information in small slices at a time but will eventually have the whole set, the desired “salami”. [28] Packet sniffer is a great tool for a hacker to compromise confidentiality by simply snooping network traffic for unencrypted data [3].

#### 4.1.2. Integrity

According to the U.S.C., integrity “*means guarding against improper information modification or destruction, and includes ensuring information nonrepudiation and authenticity*” [27]. Information has to be uncorrupted and complete in order to maintain its integrity. It is worth pointing out that information has no value to users if they cannot verify its integrity. If the information is exposed to corruption,

tampering, or even destruction, either accidental or intentional (malicious), the integrity of the information is threatened. [28]

Common attacks compromising integrity include cookie poisoning, cross-site scripting (XSS), and several kinds of message tampering or forgery [29]. Database access and modification through code injections is one of the biggest threats to both information confidentiality and integrity: according to the “2011 CWE/SANS Top 25 Most Dangerous Software Errors” listing, the most critical weakness in software is the SQL injection vulnerability [30]. Additionally, many viruses and worms spread in the Internet are explicitly designed for corrupting data [28].

One way to assure information integrity is to use hash values for files transmitted via networks. Hashing is used to check that the contents of a received file are exactly the same as the original one’s. Data corrupted during transmission, for example due to noisy channels, can be corrected with error-correcting codes, filters, and redundancy bits. [28]

#### **4.1.3. Availability**

According to the U.S.C., availability “*means ensuring timely and reliable access to and use of information*” [27]. In other words, availability enables authorized users to access the information resources when needed, without interference, and in the required format [28].

The most frequent cyber attack against information availability is denial-of-service (DoS) attack [29]. In DoS attack, the attacker sends a large number of requests (flooding) to the target system to make it crash or at least become unable to serve the legitimate requests. In distributed denial-of-service attack (DDoS), the attacker may first compromise hundreds of computers and use them to conduct a coordinated, collective attack against the target machine. These types of attacks are very difficult to defend against. [28]

### **4.2. Compromising information security with packet sniffers**

Network attacks and intrusions compromise the confidentiality, integrity, and availability of a network, computer systems in it, and information transmitted and stored in it. Packet sniffers and analyzers are utilized in several different network attacks. One of the most apparent ways for an attacker to benefit from a sniffer is literally just sniffing network traffic for some sensitive data represented in plaintext. The kind of sensitive data the attacker might be interested in includes usernames, passwords, credit card numbers, emails, session keys, and such. Furthermore, sniffers can be used even against ciphered protocols [31], and some packet analyzers include specific features for cracking encrypted passwords and decrypting hash values (see Section 3.3).

In the following, some common network attacks that utilize a packet sniffer in the attacking process are described. Most of the mentioned attacks involve the intruder impersonating another host (client, server, or router), thus enabling the

attacker to eavesdrop communications, and even add, modify, and delete messages sent between the sender and the receiver.

#### ***4.2.1. Man-in-the-Middle attack***

Man-in-the-Middle (MITM) attack is one of the main techniques used by hackers to break into computer systems [4]. In a MITM attack the intruder intercepts a communication between two hosts, eavesdrops the sent messages, and forwards them to the destination host. This gives the intruder the ability to modify and delete messages, or even insert new ones without either of the victims noticing it. The attacker splits the original connection (e.g., between a client and a server) into two separate connections: one between the attacker and host A, and the other between the attacker and host B, as shown in Figure 7.

SSL/TLS [32] (Secure Sockets Layer/Transport Layer Security) is a cryptographic protocol developed to provide secure communications over networks, especially for HTTP (Hypertext Transfer Protocol), the foundation protocol of the World Wide Web (WWW). HTTPS (HTTP Secure) is practically not a protocol, but a technique to layer HTTP on top of TLS to provide encryption and authentication for network communications [33].

However, the MITM attack can exploit the fact that during the establishment of a secure connection between a web server and a client, the server sends a certificate with its public key to the client. The public key is used to establish a secure session with the server and encrypt the client's requests. The attacker intercepts the original certificate and uses it to build a secure connection with the server. Then he creates a fake, self-signed certificate, which is sent to the client instead of the original, valid certificate, to build a separate secure connection with the client. At the end of the attack, the client and the server have an illusion of a secure communication channel between them, although in reality the attacker has the ability to capture and decrypt every message they send to each other. [31, 4]

All of the attacks described afterwards involve a Man-in-the-Middle scenario at some point, and spoofing attacks are in fact special cases of MITM attacks.

#### ***4.2.2. ARP spoofing***

Address Resolution Protocol (ARP) provides a mapping between network layer IP addresses (virtual) and link layer MAC addresses (physical). When a host wants to send an IP datagram to another host, it needs to get the MAC address of the next node on the path to the destination. If the receiver resides in the same subnet as the sender, the next node is the destination host, otherwise, the next node is a gateway/router. First the local ARP cache is searched for an entry associating the IP and MAC address of the next node. If such entry is not found, an ARP request is broadcasted to every host on the network segment. The request contains the IP address of the next node, and the machine having the specified IP responds to the sender (in unicast mode) with ARP reply, which defines the

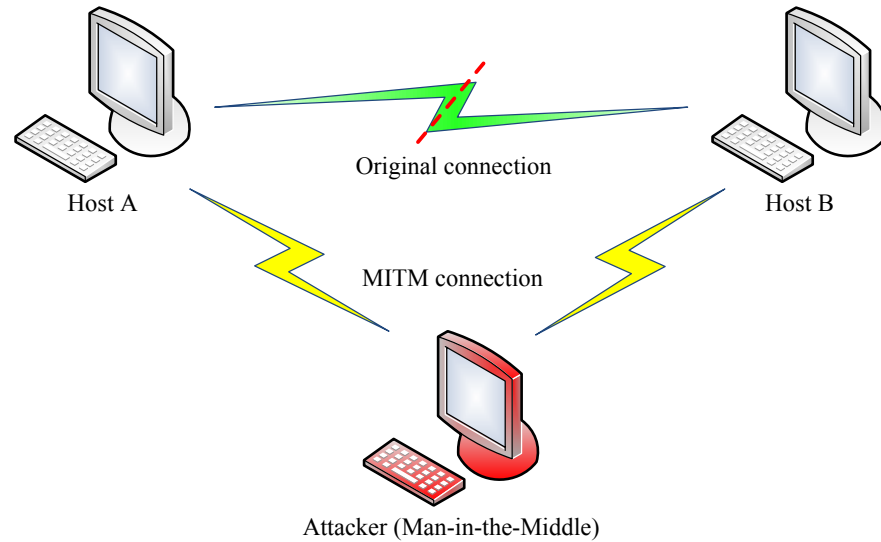


Figure 7. Man-in-the-Middle (MITM) attack.

requested IP-to-MAC address pair. After receiving the reply, the sender updates its ARP cache accordingly and sends the IP datagram encapsulated in Ethernet frame to the next node. [3]

ARP is a stateless protocol, which means that anyone can send an ARP reply to another host, even if no ARP request was sent, and the host will update its ARP cache/table with the new value. ARP request can also be used to update an existing IP-to-MAC address mapping in the cache. These possibilities make the protocol vulnerable to ARP spoofing (also known as ARP poisoning, ARP cache poisoning, and ARP poison routing). In this attack, the hacker impersonates another host, server, or gateway/router, by making the victim change the MAC address of a specific IP in its ARP cache to the hacker's MAC address. After a successful spoof, or "poisoning", all the traffic destined to the impersonated machine will be sent to the attacker's machine. [12, 4]

A sniffer can be utilized in many stages of an ARP spoofing attack. It can be used to collect information about hosts and their addresses (both IP and MAC) in the network before the actual attack. Some sniffers, for instance Cain & Abel [16], have built-in features for performing the ARP cache poisoning. Finally, after the spoofing process, a sniffer can naturally be used to analyze and examine the exchanged packets in detail.

#### ***4.2.3. DNS spoofing***

The Domain Name System (DNS) is a vast, hierarchical collection of distributed name servers that provides a mapping between domain names (e.g., `www.example.com`) and IP addresses (e.g., `192.168.0.1`). Domain names are used because they are easier for humans to remember and use, but they have to be resolved into IP addresses used by computers to communicate with each other.

When a user wants to access a particular website or other web resource, he enters the domain name into a web application (e.g., a web browser or an email client). The user's computer will then search its local cache for the corresponding resource record (RR), and if it does not find a match, it sends a DNS query to the nearest DNS server. The server also checks first its cache, but if still no match is found it will forward the query to the root DNS server, and the Domain Name System will perform the name resolution (iteratively or recursively). Eventually, the local DNS server receives a reply containing the requested resource record, forwards it to the client, and the client connects to the given IP address. [34]

In DNS spoofing, or DNS cache poisoning, the attacker pretends to be an authoritative server higher in the hierarchy and replies to the DNS query made by the inferior server. If the requesting server does not authenticate the response, it will end up caching an incorrect domain name to IP address mapping, in other words, the attacker “poisons” the cache of the DNS server. Thus the unsuspecting user can be directed to a malicious server controlled by the attacker instead of the trusted one. The user may then be lured to use a fake website (e.g., similar to an official online banking website) and enter passwords or other sensitive information. [34]

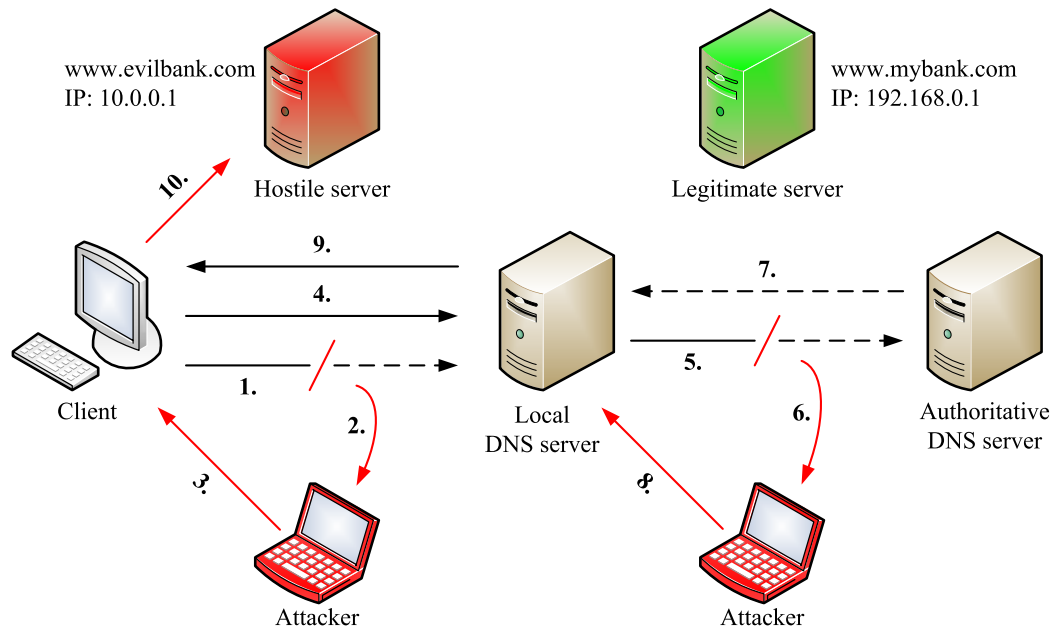
There are different kinds of DNS spoofing attacks and various terms are used for the same attack. For example, in a different DNS spoofing attack, also known as DNS ID spoofing [35], the attacker conducts first an ARP cache poisoning, sniffs the DNS identification number that binds the query and the response together, and sends a spoofed DNS response to the victim. This technique does not corrupt any DNS server's cache and thus affects only one host. The term “pharming” (a neologism based on the words “farming” and “phishing”) is also used for attacks that exploit poisoned DNS servers and redirect network traffic to bogus websites created by the hacker [35]. DNS spoofing can be performed very easily with packet sniffers, such as Ettercap [18] and Cain & Abel [16]. Both DNS ID spoofing and server-side DNS cache poisoning are illustrated in Figure 8.

#### *4.2.4. Session hijacking*

Session hijacking attacks are used to take over already existing connections. An intruder can bypass strong authentication schemes, such as one-time passwords, by hijacking the connection after the authentication process is completed. Sniffers are basic tools used in conducting these kinds of attacks. [3]

Hypertext Transfer Protocol (HTTP) is a stateless protocol so web-based applications often use cookies to maintain their state. Without a way to preserve the state of the session, online shopping and banking would be impossible. Cookies are header fields inside HTTP requests and responses exchanged between a client and a server. The information in these fields can be, for example, a user identifier, a database key, or a session ID. [36]

A session ID (identifier) is a piece of data that is used in a communication between a client and a server to uniquely identify a session. A session, on the other hand, can be thought of as the interval between when the server starts keeping track of the client, typically a browser, and when it stops. A session ID is



#### DNS ID spoofing

1. Client sends a DNS query to the local DNS server, asking for the IP address of the domain name *www.mybank.com*.
2. Attacker captures the query before the server receives it.
3. Attacker sends a spoofed DNS response to the client, saying that the IP address of *www.mybank.com* is 10.0.0.1, and thus poisoning the DNS cache of the client.

#### DNS cache poisoning (server-side)

4. Client sends a DNS query (as in 1.) to the local DNS server without interruption.
5. Local DNS server forwards the query to the next server higher in the DNS hierarchy.
6. Attacker captures the forwarded query, thus interrupting the valid name resolution.
7. Authentic response from the Domain Name System hierarchy is never received.
8. Attacker sends a spoofed DNS response (as in 3.) to the server, thus poisoning its DNS cache.
9. Local DNS server forwards the spoofed DNS response to the client (and everyone else requesting for it afterwards).
10. As the result from both methods, the client connects to the IP address 10.0.0.1, which is a server controlled by the attacker, instead of the legitimate one at 192.168.0.1.

Figure 8. Two different DNS spoofing attacks.

usually some pseudo-random number/string or hash value. It can be exchanged in many ways: including it in an URL, a hidden field, or a cookie [37].

Session hijacking is a very popular and easy way to hack into other people's webmails (web-based email, accessed via browser). The attacker can use a sniffer to read the sent messages in a specific session, capture cookies, and reveal the used session ID. After the attacker has obtained the victim's session ID, he will use some cookie editor to substitute his own session ID with the stolen one. By sending the modified cookie inside a request to the server, the attacker will have access to the authenticated session, be it a webmail, an online banking, or an online shopping session. [37, 12]

Firesheep<sup>1</sup> is an extension for the Firefox web browser. Basically, it is a packet sniffer, which grabs unencrypted cookies over an open wireless network, and allows the user to hijack someone's session. Its operation is based on the fact that many websites encrypt only the login process, not the whole session. Thus the cookies and the session ID are easily sniffed. Another great software for session hijacking is the Sidejacking tool set<sup>2</sup>, developed by Errata Security for network security testing. It consists of Ferret, a tool for detecting data seepage, and Hamster, a tool for performing the actual session hijacking (sidejacking).

#### 4.2.5. *Replay attack*

In a replay attack the attacker sniffs a conversation between two hosts and retransmits a captured message or a message sequence. The attacker does not have to produce any messages himself or even read the captured messages in cleartext. This means that he can, for example, catch a password hash transmitted during an authentication process and then reuse it as a fake proof of identity after the valid session is over. [38]

Data used in an encrypted session should not be valid in another session. To mitigate replay attacks, certain freshness identifiers can be used, such as timestamps, nonces (arbitrary numbers used only once), session IDs, and one-time passwords.

### 4.3. Protecting information security with packet sniffers

Packet sniffers were not originally designed for breaking into systems, eavesdropping on communications, or any other malicious activities but instead for testing systems and implementations in order to find flaws and detect possible vulnerabilities before they are exploited. Sniffers are also very handy for administrators to monitor the network usage and troubleshoot occurring problems. Some of the analyzers available have a command-line interface, which makes them usable via scripts, and thus easy to utilize for different purposes.

#### 4.3.1. *Network troubleshooting*

Packet analyzers are used to solve, or at least clarify, many different network problems. Typical examples of such situations are: a lost TCP connection, an unreachable destination or port, problems with packet fragmentation, the lack of network connectivity, and malware related issues. [1, Ch. 7]

For example, an ICMP (Internet Control Message Protocol) type 8 packet (Echo request) is used to *ping* (used to test the reachability of a host) the destination, and a type 0 packet (Echo reply) is expected as a response on success.

---

<sup>1</sup><http://codebutler.com/firesheep>

<sup>2</sup><http://www.erratasec.com/erratasec.zip>

However, in the case of an unreachable destination, a type 3 packet (Destination unreachable) is received. With the help of a packet analyzer, the response packet can be examined in greater detail, and more useful information can be gained from the Code field of the ICMP packet. The Code field contains another number that corresponds to a more accurate description of the message, such as 0 (Destination network unreachable), which would mean that the whole network where the destination host is located in is currently unavailable. [1, Ch. 7]

One of the most common network problems is the loss of network connectivity. TCP is designed so that if a packet is sent to a destination, but no reply is received from it, the original packet is retransmitted after a certain amount of time. Nonetheless, too many TCP retransmissions are usually a sign of a connectivity problem. By being able to locate the exact packet at which the TCP retransmission attempt begins, the network administrator may figure out why the loss of connectivity occurred. [1, Ch. 7]

#### ***4.3.2. Packet filtering***

A packet analyzer can be configured to filter suspicious packets from the network traffic according to specified rules. The packet filtering functionality of sniffers is similar to the first generation firewalls, known as packet filtering or network layer firewalls. Both can filter network packets based on the source/destination IP address and port number, used protocol, timestamp, and many other attributes. [28, p. 241-244]

Packet filtering is a simple technique to improve network security, at least a little bit. However, it can be extended easily by running the sniffer inside a script, and then executing some supplementary actions based on the recognized, possibly malicious packets.

#### ***4.3.3. Intrusion detection***

Intrusion detection system (IDS) is a device or a software application that monitors the network in order to detect intrusions and other malicious activities. To be exact, it does not detect intrusions but identifies evidence of intrusions, either while they are still in progress or after the incidents [39]. After detecting an intrusion the IDS typically logs information about the incident and notifies the network administrator.

The intrusion detection system uses a packet sniffer to read and analyze network traffic. By examining both the packet header fields and the packet contents, IDSs are able to detect several types of attacks, such as DoS and spoofing attacks. Some of the automated responses may even try to stop the intrusion by shutting the attacked system down, or closing open sessions and connections. [40]

Intrusion detection techniques can be divided into two basic categories: anomaly based detection and misuse/signature based detection. Anomaly based detection uses models to define the correct behavior of users and applications, thus interpreting the deviant behavior as alarming. Misuse based detection, on the



contrary, uses attack patterns, “signatures”, to define the wrong behavior. The signatures are matched against the monitored data streams to find evidence of known attacks. [39]

IDS tools are a general part of security solutions nowadays. Snort<sup>3</sup>, for example, is an open source network intrusion detection and prevention system (IDS/IPS) developed by Sourcefire. It combines the benefits of both anomaly and signature based detection techniques, and it is able to detect ARP cache poisoning attacks among other things. [41]

#### ***4.3.4. Data leakage detection***

In data leakage incidents, sensitive data is disclosed to unauthorized personnel either by a malicious action or an unintentional error. Emails, instant messaging, file exchanges, e-commerce, and other online services used by employees and individuals are all potential threats to the confidentiality of sensitive information. A packet analyzer can be used to analyze both inbound and outbound network traffic to detect sensitive data that is being transmitted in violation of information security policies.

Data leakage/loss prevention (DLP) system aims to protect data at three levels: data-at-rest (in static storage), data-in-use (in local processing), and data-in-motion (in network traffic). Regular expressions, keywords, and other pattern matching techniques suit well for identifying structured data (e.g., credit card numbers, social security numbers, and medical records). For less structured data, DLPs use partial document matching and hash fingerprinting. Hart et al. propose automatic document classification algorithms for identifying data as either sensitive or non-sensitive. Their method is scalable and it is able to detect over 97 % of data leakages. [42]

---

<sup>3</sup><http://www.snort.org/>

## 5. WIRESHARK DISSECTORS

One of the key strengths of Wireshark [5] is that users can extend it to analyze their own protocols by writing custom dissectors for it. Dissectors are small pieces of software meant to dissect the captured data stream into separate packets and fields, and analyze them according to the specified protocol rules. They are used to display the packet data in a readable format in Wireshark’s user interface, instead of the binary or the hexadecimal representation. Each protocol has its own dissector, so usually several consecutive dissectors are needed to dissect a complete packet. Each dissector analyzes its own header fields from the packet and then hands the payload part to the next dissector higher in the protocol stack. Wireshark uses static routes and heuristics in trying to find the correct dissector for each protocol.

Wireshark dissectors can be written in either C or Lua. Both alternatives are described and compared briefly below, and in Section 5.3 is an introduction to dissector generators, the empirical part of this thesis. At the end of the chapter, some related works are also presented.

### 5.1. Dissectors written in C

C [43] is a procedural, statically typed, and relatively low-level programming language. Wireshark source code is written in C, and so are all the built-in protocol dissectors. There are two ways to create custom dissectors for Wireshark: either as standard dissectors or as plugins. C is often preferred in writing dissectors since it has shorter execution times. However, there are some drawbacks concerning the development phase. Wireshark has to be recompiled every time a non-plugin dissector is added or modified, which is a very time-consuming task. Even adding a dissector as a plugin is a tedious task since about 10 new files have to be created to the plugin folder along with the actual source files of the dissector, and on top of that, almost as many existing Wireshark files have to be modified [44].

### 5.2. Dissectors written in Lua

Lua [45] is a lightweight, dynamically typed, multi-paradigm programming language designed as a scripting language [46]. Lua dissectors are very similar to those written in C. Both C and Lua dissectors have to be registered to be able to handle a packet or a payload that fulfills the specified protocol rules. As a scripting language, and without the need to recompile Wireshark over and over again, or create and modify a bunch of extra files, Lua is ideal for rapid prototyping and testing of new protocol dissectors. It is also much more user-friendly and less error-prone than C due to its automatic memory management with incremental garbage collection. Even though Lua is not as fast as C, it is fast. In fact, Lua has deserved reputation for performance, and it is currently the leading scripting language used in games [46, 47].

A sample protocol dissector written in Lua is presented in Figure 9. On line 2, a new dissector named *myproto* is created. On lines 5–11, all the protocol fields are defined, see the detailed description of *ProtoField* objects later. Line 5 defines the fields table of the dissector and line 6 defines the *valuestring* table referenced on line 10. Between lines 14 and 34, is the actual protocol dissector function, the detailed description follows afterwards. On line 37, an existing dissector table is loaded, in this case for UDP (User Datagram Protocol), and on the last line, the created dissector is added to the table to be used for packets that arrive from port 1000. Adding the dissector into a dissector table is called registering the dissector.

```

01 -- Create a new dissector
02 MYPROTO = Proto("myproto", "My Simple Protocol")
03
04 -- Create the protocol fields
05 local f = MYPROTO.fields
06 local formats = {"Text", "Binary", [10] = "Special"}
07
08 f.msgid = ProtoField.uint32("myproto.msgid", "Message Id")
09 f.magic = ProtoField.uint8("myproto.magic", "Magic", base_HEX, nil, 0xF0)
10 f.format = ProtoField.uint8("myproto.format", "Format", nil, formats, 0x0F)
11 f.mydata = ProtoField.bytes("myproto.mydata", "Data")
12
13 -- The dissector function
14 function MYPROTO.dissector(buffer, pinfo, tree)
15
16     -- Add fields to the tree
17     local subtree = tree:add(MYPROTO, buffer())
18     local offset = 0
19     local msgid = buffer(offset, 4)
20
21     -- Modify pinfo columns
22     pinfo.cols.protocol = MYPROTO.name
23     pinfo.cols.info = "Message Id: "
24     pinfo.cols.info:append(msgid:uint())
25
26     subtree:add(f.msgid, msgid)
27     subtree:append_text(", Message Id: " .. msgid:uint())
28     offset = offset + 4
29     subtree:add(f.magic, buffer(offset, 1))
30     subtree:add(f.format, buffer(offset, 1))
31     offset = offset + 1
32     subtree:add(f.mydata, buffer(offset))
33
34 end
35
36 -- Register the dissector
37 udp_table = DissectorTable.get("udp.port")
38 udp_table:add(1000, MYPROTO)

```

Figure 9. Sample protocol dissector written in Lua.

## ProtoField

*ProtoField*<sup>1</sup> objects are used to specify the fields of the protocol. The objects are defined as *ProtoField*.<type>(abbr, [name], [base], [valuestring], [mask], [desc]), in which the <type> denotes the type of the protocol field (e.g., float, bool, bytes, or string), and the arguments are

- *abbr* – The abbreviated name of the field.
- *name* (optional) – The actual name of the field.
- *base* (optional) – The base of the field (e.g., base\_DEC or base\_HEX).
- *valuestring* (optional) – The table containing the corresponding text values for the numerical values.
- *mask* (optional) – The binary mask of the field.
- *desc* (optional) – The description of the field.

## Dissector function

The dissector function takes the following three arguments as input

- *buffer* – The packet data to be dissected is held in this buffer.
- *pinfo* – The packet info structure which contains general information about the protocol.
- *tree* – The protocol tree structure to which the dissected protocol items are added.

On lines 17–19, a subtree is built for the dissection results, offset (used in reading the buffer) is initialized to zero, and the first four bytes of a packet/payload are assigned to *msgid* variable. On lines 22–24, the name of the protocol is set to the protocol column of the *pinfo* structure and the message ID, the value of the *msgid* variable, is set to the info column.

On lines 26 and 27, the *msgid* value is set to the *f.msgid* protocol field and added to the tree along with a short textual declaration. Next, on lines 28–30, the offset value is increased by four, and consequently the fifth byte is assigned to both *f.magic* and *f.format* field of the protocol tree. Since one byte is assigned to two fields, the binary masks defined at the end of lines 9 and 10 are now used. This results in the higher nibble (four bits) of the byte being assigned to *f.magic* and the lower nibble to *f.format*. On line 31, the offset is incremented by one, and on the last line of the dissector function, the rest of the buffer is read and assigned as the packet data.

An example result of dissecting a packet of *myproto* protocol with the presented Lua dissector can be seen in the screenshot of Wireshark in Figure 10.

---

<sup>1</sup>[http://www.wireshark.org/docs/wsug\\_html\\_chunked/lua\\_module\\_Proto.html](http://www.wireshark.org/docs/wsug_html_chunked/lua_module_Proto.html)

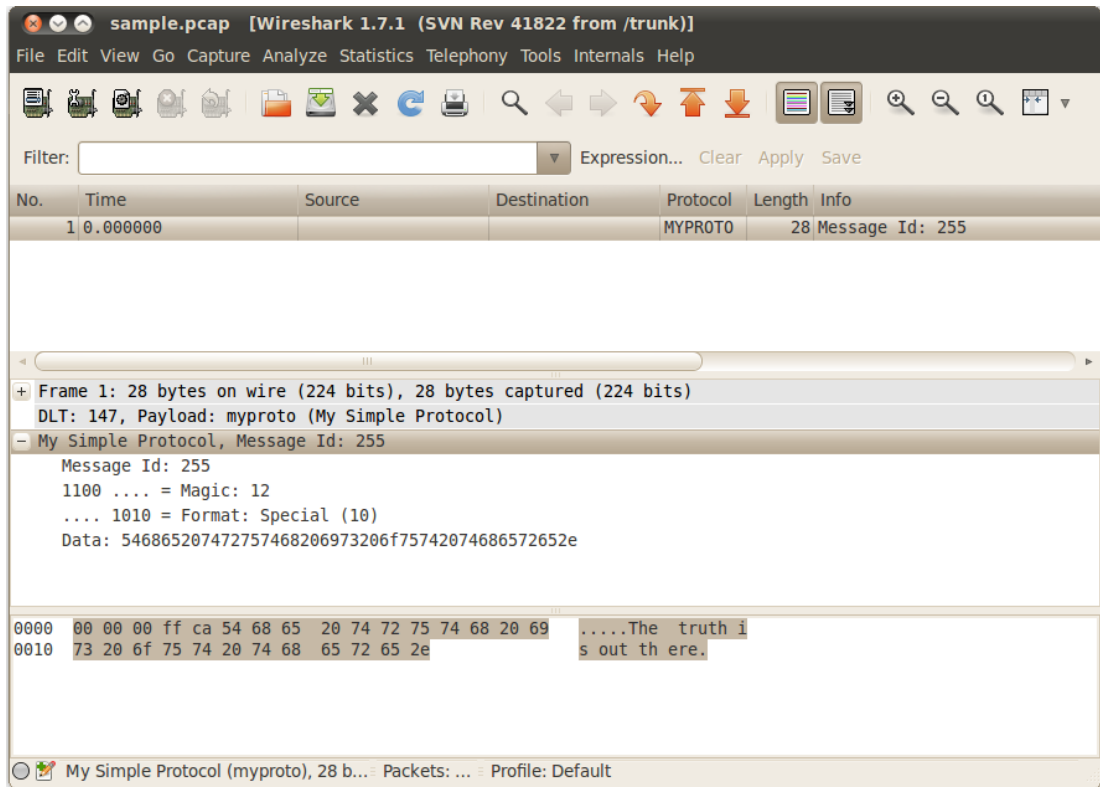


Figure 10. The result of dissecting a *myproto* packet with Wireshark.

## DissectorTable

*DissectorTable* is a table of subdissectors of a particular protocol, used to handle the payload. For example, UDP subdissectors are added, or registered, to the *udp.port* table. New dissector tables can also be created with the *new* function of the *DissectorTable* class.

### 5.3. Dissector generators

Dissector generators are exactly what the name implies: tools that generate dissectors. Their purpose is to ease and speed up the creation of dissectors. They include automation at some level; for example, some tools can be fed different input files according to which the dissectors are generated automatically, and some can be given the rules of the protocol as a formal language or by defining them through some kind of graphical interface, as in the case of this thesis. Next, some related works in this field are presented briefly.

### 5.3.1. *CSjark*

CSjark<sup>2</sup> is a tool for generating Wireshark dissectors written in Lua from C struct definitions. It was developed by seven Norwegian students as a course project at the Norwegian University of Technology and Science. The project was executed on behalf of Thales Norway AS<sup>3</sup>, which is an international electronics and systems group, focusing on defense, aerospace, and security markets. [48]

CSjark is implemented in Python and it makes use of some open source libraries and tools, such as pycparser<sup>4</sup>, Python Lex-Yacc (PLY)<sup>5</sup>, and PyYAML<sup>6</sup>. The tool is used through a command-line interface. It uses an external C preprocessor for processing the header files where the C structs are defined. Then the C code is given to pycparser, which parses the code and generates an abstract syntax tree. CSjark traverses the tree and creates a new protocol for every struct it finds. PyYAML is used to parse the given configuration file, which specifies the options used in the generation of the dissectors. The last step is that CSjark creates a Wireshark dissector written in Lua for every protocol derived from a C struct.

### 5.3.2. *Interpreted dissector*

In [49], a special, interpreted dissector is presented. It was developed on behalf of Ericsson to help them develop and debug their telecommunication router. The program reads a formal protocol definition written in a text file and uses that information to build/rebuild the dissector to be able to present the captured network traffic in the right way. It uses a tool called BNFC<sup>7</sup> (Backus-Naur Form Converter), which reads a file containing a specification of the language in formal language and generates a lexer, a parser, and skeleton code with an abstract syntax tree. The protocol specification language itself uses C-like declarations.

The motivation to develop the tool was that Ericsson uses proprietary protocols that cannot be disclosed to the public and the protocols change often thus requiring frequent modifications to the dissectors [49]. Instead of recompiling Wireshark or making new plugins every time a protocol changes, only the contents of a text file have to be edited and Wireshark restarted. Most importantly, this makes changing and debugging the protocol implementations easier and faster. Additionally, it simplifies maintaining proprietary protocols, since the actual protocol definition file does not have to be under the GPL<sup>8</sup> (GNU General Public License), even though Wireshark's source code is.

---

<sup>2</sup><http://readthedocs.org/docs/csjark/en/latest/>

<sup>3</sup><http://www.thales.no/>

<sup>4</sup><http://code.google.com/p/pycparser/>

<sup>5</sup><http://www.dabeaz.com/ply/>

<sup>6</sup><http://pyyaml.org/>

<sup>7</sup><http://www.cse.chalmers.se/edu/year/2011/course/TIN321/lectures/bnfc-tutorial.html>

<sup>8</sup><http://www.gnu.org/licenses/gpl.html>

## 6. LUA DISSECTOR GENERATOR

In this chapter, the empirical part of the thesis is described. The tool, Lua Dissector Generator (LuDis), was developed in order to make it possible for people with little or no experience in programming to create protocol dissectors and to make the creating of dissectors for new protocols faster and easier.

Almost every nook and cranny of the world is connected to each other through a huge network of networks. To be able to solve network problems and protect our networks from malware and intrusions, dissectors are needed to make protocols and their packets understandable. As said by a security researcher Chris Sanders, “*All network problems stem from the packet level, where even the prettiest-looking applications can reveal their horrible implementations and seemingly trustworthy protocols can prove malicious.*” [1]. Dissectors are the key to expose these implementations, help in discovering possible vulnerabilities in them, and to detect malicious packets in the network.

### 6.1. Functionality and logic

The principal functionality of LuDis is to generate Wireshark dissectors in Lua according to the protocol rules defined by the user. The rules are specified basically by opening a sample packet of the protocol with the tool and defining all fields of the packet. A packet can contain several fields and subfields, but they must not overlap. The usage of the tool is explained in detail in Section 6.3.

The main logic of Lua Dissector Generator is illustrated in Figure 11. The figure contains only the most important activities and thus is not a thorough description of the tool. The logic of defining a new protocol field is explained in a separate diagram in Figure 12. In other words, Figure 12 illustrates the inner logic of the activity marked with *A* in Figure 11. In the following, the program logic presented by these activity diagrams is explained in more detail.

After the program is started, the main window of LuDis is shown to the user and the program enters *Ready* state. By pressing the *Help* button the user is given instructions on how to use the program. At this point, most of the buttons lead to an info popup that advises to open a sample packet. In fact, info and error popups are utilized throughout the program to help and inform the user. The *Open* button must be pressed to proceed to the first step in the dissector generation process. This action will present the user a file dialog for selecting an input file that contains a sample packet of the target protocol. The input file is checked for validity (see Section 6.3.2 for details) and if it is invalid, an error is thrown, otherwise the sample packet is displayed in the main window.

The next step is to define the information concerning the protocol itself. It is done by clicking the *Define protocol* button and filling the necessary information in the opened window. If the filled protocol information is accepted, LuDis creates a protocol field tree for the protocol and displays it in the main window below the sample packet. Now the user may begin defining the fields and subfields of the packet.

The process of defining (and deleting) protocol fields/subfields proceeds as depicted in Figure 12. First the type of the field delimitation must be selected. In the case of fixed length fields, the bytes of the field to be defined must be highlighted in the sample packet with mouse. In the case of using delimiters to define field lengths, the delimiters are specified with the window which opens after pressing the *Define delimiters* button, and the field to be defined is selected with the arrow buttons. In both cases, after selecting the bytes, the next step is the actual field definition by pressing either the *Integer type field* or the *Other type field* button. The necessary information is specified with the opened window, the entered data is checked, and if everything is in order, a new protocol field is created and added to the protocol tree.

Deleting a field/subfield is similar in both cases. First the field to be deleted is selected from the protocol tree by highlighting it and clicking the highlighted area. Next the *Clear defined field* button is pressed and the tool checks if the selected field has any subfields. If it does, the user is warned because also the subfields of the selected field will be deleted. Eventually, the defined field information is deleted (along with its subfields) and removed from the protocol tree.

The protocol tree can be saved into a file or deleted at any time with the *Save* or *Clear protocol tree* button, respectively. The final step is to press the *Create dissector* button, define the remaining information, and command the tool to generate the protocol dissector. The dissector file is created into the specified directory with a filename formed from the protocol name and “.lua” suffix.



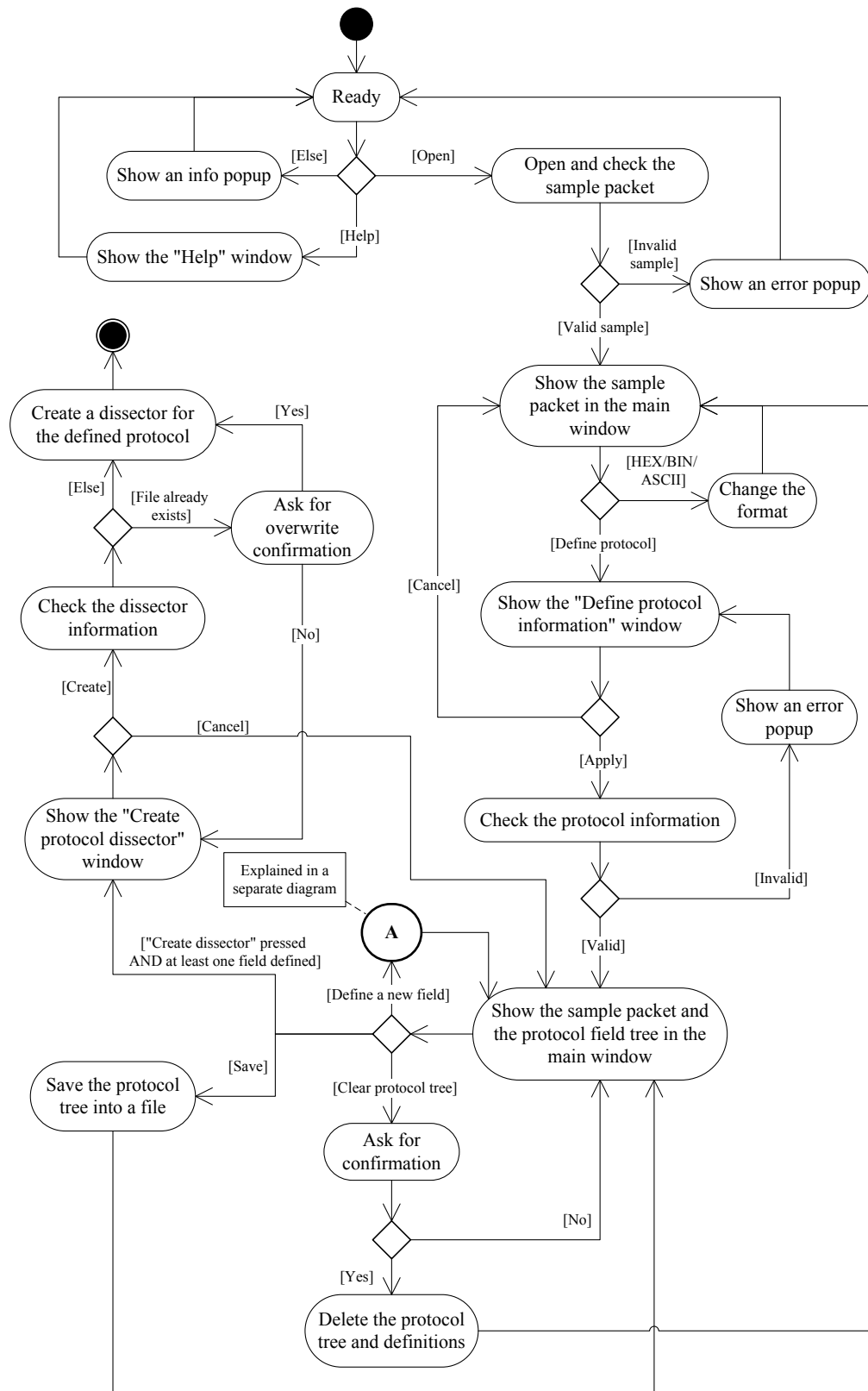


Figure 11. The main logic of Lua Dissector Generator.

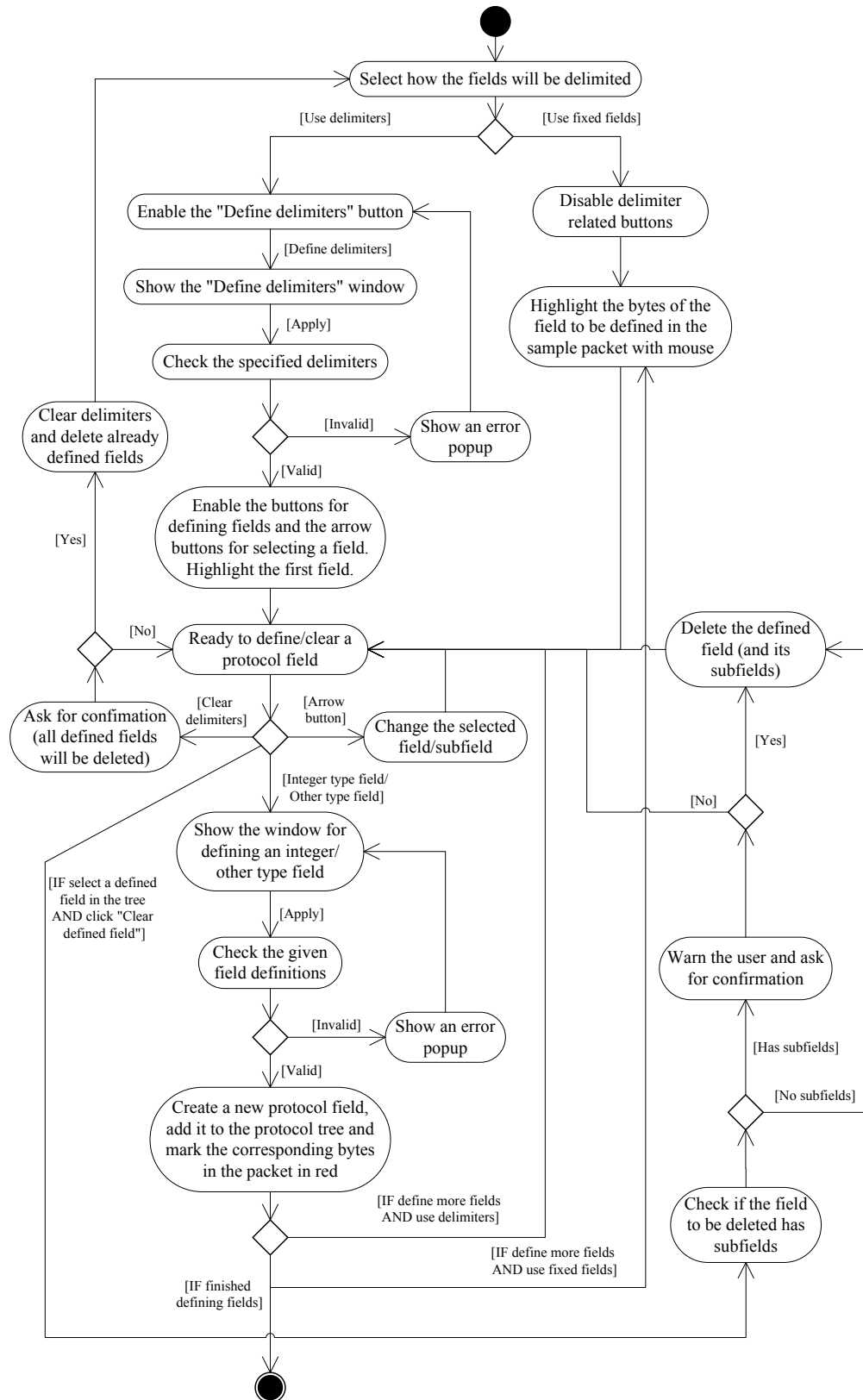


Figure 12. The logic of defining/deleting a protocol field/subfield.

## 6.2. Implementation

In the following subsections, some decisions made regarding the development process of the tool are explained and justified.

### 6.2.1. *Programming language for the tool*

All the source code of LuDis is written in Python [50] programming language. Python is a multi-paradigm, dynamically typed, high-level programming language. It is often used as a scripting language because it offers dynamic typing and automatic memory management, and has no need for compilation process since it is an interpreted language. These features make it also ideal for rapid application development and prototyping. For these reasons it was chosen as the implementation language of this tool.

### 6.2.2. *Programming language for generating the dissectors*

LuDis generates the protocol dissectors in Lua. Lua, as described in Section 5.2 of Chapter 5, is suitable for prototyping and experimenting with custom dissectors very much for the same reasons as Python is a good choice for programming. After some consideration, Lua was chosen as the programming language of the dissectors instead of C, the other officially supported language for writing Wireshark dissectors. Python was rejected too since the support for it is still a work in progress [51] and there is little help available on the Internet. Although the author had some previous experience with C but none with Lua, it was considered as a viable option.

### 6.2.3. *Development environment and tools*

The software was developed on Ubuntu 10.04 (Lucid Lynx) operating system and the version of the Python interpreter used was 2.6.5. All the source code was written with Sublime Text 2 text editor. Diagrams and drawn figures were produced with Microsoft Office Visio 2007. The thesis document itself was written in L<sup>A</sup>T<sub>E</sub>X typesetting language with TeXnicCenter, which is an integrated environment for creating L<sup>A</sup>T<sub>E</sub>X documents on Windows.

### **Tkinter GUI toolkit**

Tkinter<sup>1</sup> is the de facto standard for developing portable graphical user interfaces in Python. It is regarded as a simple and lightweight open source GUI (graphical user interface) toolkit. It was chosen to be used in the implementation of the tool because of its accessibility, portability, and availability. In other words, Tkinter is easy to get started with to build simple GUIs; a Python script that

---

<sup>1</sup><http://www.pythonware.com/library/tkinter/introduction/>

builds a GUI with Tkinter will work without modifications on all major platforms (Microsoft Windows, Unix/Linux, and OS X); and Tkinter is a standard module in the Python library, meaning it should be bundled with the Python interpreter installation. Tkinter has also a good documentation and manuals available both in the literature and online. [52, Ch. 6]

### 6.3. User interface and usage

The following subsections describe the usage of the tool in detail, beginning from getting the program running and finishing to utilizing the generated dissectors in Wireshark. All the buttons and text fields of different windows are explained with pictures and usage examples are given when possible.

#### 6.3.1. *How to run the software?*

The program was developed and tested on Ubuntu 10.04, but it should work without problems on most Linux and UNIX based operating systems. LuDis was tested also on Windows Vista and Windows 7, and it worked fine, but the GUI looks slightly different on Windows and graphical proportions are not the same as on Linux. For that reason, Ubuntu is strongly recommended as the operating system when using the tool.

Since Python code does not need to be compiled, but interpreted instead, a Python interpreter<sup>2</sup> is needed. The current version of LuDis does not support Python interpreter version 3, so the newest release of version 2 is recommended. Tkinter toolkit should be bundled with the interpreter installation. After successfully installing a Python interpreter, the program can be started from the command line by going to the source code directory and entering `python ludis.py` command. Alternatively, by giving the main file, namely `ludis.py`, execute permission the program can be run simply with `./ludis.py` command (in Linux/UNIX environments).

#### 6.3.2. *Input files*

Input files must be in standard ASCII<sup>3</sup> (American Standard Code for Information Interchange) format. Also non-printable control characters are allowed. In addition, the content of an input file has to follow certain rules. First of all, the tool accepts three types of content presentations: binary, hexadecimal, and ASCII. The first three characters of the input file must express one of these valid types. Rules for each content presentation are the following

1. *Binary presentation* – The input file must begin with “BIN” (case-sensitive) otherwise the file is rejected and an error popup is shown to the user ex-

---

<sup>2</sup><http://www.python.org/download/releases/>

<sup>3</sup><http://www.asciitable.com/>

plaining the reason. After the type declaration follows the actual sample packet of the protocol. Accepted characters for this data are 0 and 1, which represent bits. All whitespaces are ignored. The packet must consist of full bytes of data, in other words, the packet length must be evenly divisible with eight (whitespaces ignored). This is because one byte represents eight bits.

2. *Hexadecimal presentation* – The input file must begin with “HEX” (case-sensitive) otherwise the file is rejected and an error popup is shown to the user explaining the reason. After the type declaration follows the actual sample packet of the protocol. Accepted characters for this data are digits 0–9 and letters A–F (case-insensitive), which represent hexadecimal values. All whitespaces are ignored. The packet must consist of full bytes of data, in other words, the packet length must be evenly divisible with two (whitespaces ignored). This is because one hexadecimal digit represents one nibble, which is half a byte.
3. *ASCII presentation* – The input file must begin with “ASC” (case-sensitive) otherwise the file is rejected and an error popup is shown to the user explaining the reason. After the type declaration follows the actual sample packet of the protocol. Accepted characters for this data are all standard ASCII characters (case-sensitive), including control characters. Control characters can be added to an ASCII file with escape sequences. For example, escape sequence “\n” represents ASCII line feed (LF) and “\r” ASCII carriage return (CR). In this case, all whitespaces, *except* single space characters, are ignored. The packet must consist of full bytes of data, in other words, the packet must be at least one character long. This is because one ASCII character is represented with one byte.

### 6.3.3. Main window

In Figure 13, the main window of LuDis after the startup is shown.

#### Frames

There are two text frames in the main window. The purpose of each is the following.

- *Sample packet frame* – The upper frame is used to display the sample packet read from the input file. The sample packet data is divided into pieces of full bytes, i.e., hexadecimal are displayed in sets of two characters separated by a space and similarly binaries in sets of eight digits.
- *Protocol tree frame* – The lower frame is used to display the current protocol field tree as it is being constructed. As the protocol is being defined, the name and the description of the protocol are displayed at the top of the frame. Below it come the defined protocol fields on indentation level one and their subfields on indentation level two.

## Menu buttons

The functions of the uppermost buttons are

- *Open* – Opens a file dialog for the user to select an input file, which contains the sample packet of the target protocol.
- *Save* – Gives the user the possibility to save the current protocol tree to a text file. The button is disabled when the protocol tree frame is empty.
- *Help* – Shows the usage instructions of the tool in a new window. The opening window is explained in Section 6.3.4.
- *Quit* – Exits the program, but presents first a confirmation popup to the user.

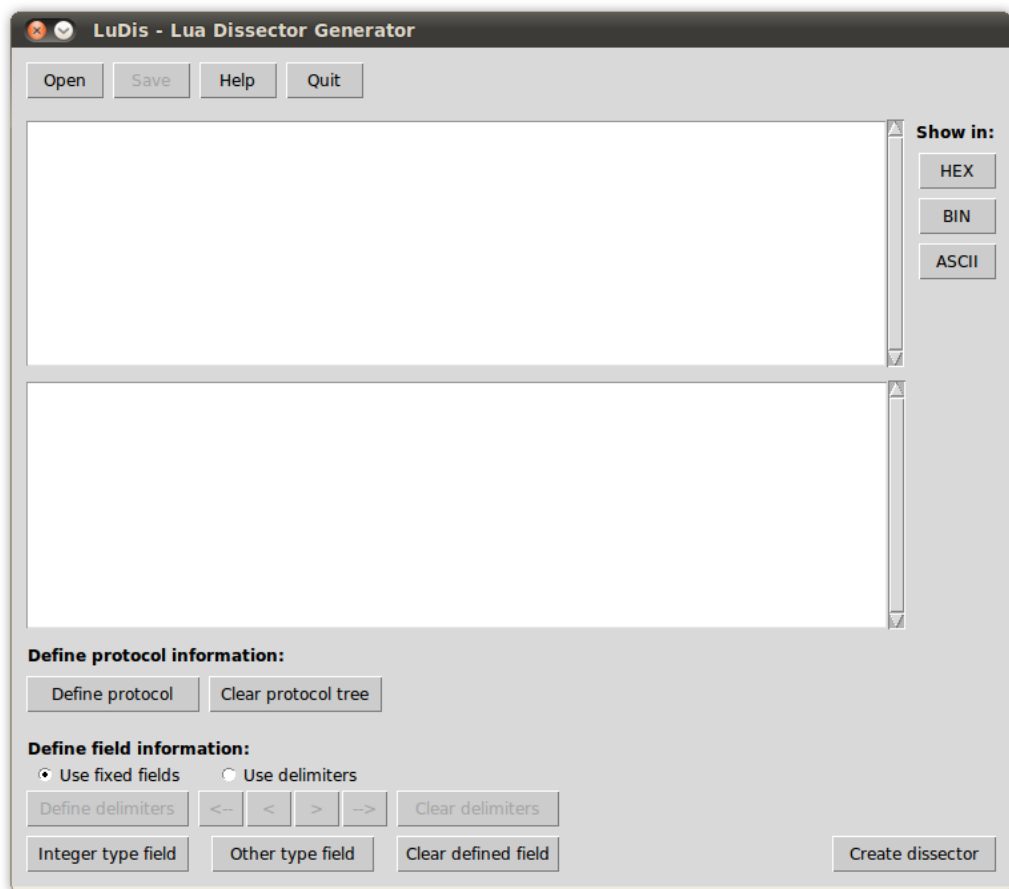


Figure 13. The main window of LuDis.

## Format buttons

The presentation of the sample packet can be switched between three different formats: hexadecimal, binary, and ASCII. The format can be changed with the buttons on the right, below the text “Show in”.

- *HEX* – Shows the sample packet in hexadecimal.
- *BIN* – Shows the sample packet in binary.
- *ASCII* – Shows the sample packet in ASCII. Non-printable control characters are replaced with dots in the packet frame.

### Define protocol information buttons

These two buttons are related to the whole protocol definition and the protocol tree.

- *Define protocol* – Opens a new window on top of the main window for defining the protocol information. The opening window is explained in Section 6.3.5.
- *Clear protocol tree* – Deletes the whole protocol field tree including all field definitions. This results also in clearing the protocol tree frame and removing red markings (fields already defined) from the sample packet. To prevent mistakes the user is asked to confirm the action before the protocol tree is cleared.

### Define field information buttons

The two radio buttons are used for choosing the way how the fields and subfields are going to be delimited in the sample packet.

- *Use fixed fields* – By choosing this option the user has to delimit the fields and subfields with mouse. The limits of a field are defined by highlighting the relevant bytes in the sample packet. The word “fixed” refers to the fact that the defined fields have to have the same size and position in each packet of the protocol, in other words, the sizes of the fields are fixed. This option also disables the delimiter related buttons.
- *Use delimiters* – By choosing this option the user has to specify certain delimiters for fields and subfields. The tool uses these delimiters to automatically determine the ends of the fields. This way the defined fields can be of varying size in separate packets. These field delimiters are explained in Section 6.3.6. This option also enables the delimiter related buttons.

Below the radio buttons there are six buttons concerning the field delimiters. They are used to define and clear the delimiters and select a field/subfield to be defined. To be able to use these buttons *Use delimiters* option must be selected.

- *Define delimiters* – Opens a new window on top of the main window for defining the delimiters for protocol fields and subfields. The opening window is explained in Section 6.3.6.

- <- – Selects and highlights with green the previous field in the sample packet. Fields are separated by field delimiters. This arrow button is not enabled until the field delimiter is defined and found inside the sample packet.
- < – Selects and highlights with yellow the previous subfield inside the selected field. Subfields are separated by subfield delimiters. This arrow button is not enabled until the subfield delimiter is defined and found inside the selected field.
- > – Selects and highlights with yellow the next subfield inside the selected field. Subfields are separated by subfield delimiters. This arrow button is not enabled until the subfield delimiter is defined and found inside the selected field.
- -> – Selects and highlights with green the next field in the sample packet. Fields are separated by field delimiters. This arrow button is not enabled until the field delimiter is defined and found inside the sample packet.
- *Clear delimiters* – Clears the defined field and subfield delimiters. If there are any defined fields in the protocol tree, the user is asked for confirmation since all the field and subfield definitions are deleted in the process.

At the bottom of the main window, there are three buttons that are used to define the fields and subfields. The field type buttons exclude each other, in other words, only one or the other can be chosen per field/subfield. Before pressing either button the type of the field content should be decided: Should the field be interpreted as integer data or as some other type of data during the packet dissection process in Wireshark? Finally, the solitary button in the lower right corner is logically the last button to be used in the dissector generation process.

- *Integer type field* – Opens a new window on top of the main window for defining the field/subfield selected (highlighted) in the sample packet. If this button is pressed, the content of the field/subfield will be interpreted as integer data during the packet dissection process in Wireshark. The opening window is explained in Section 6.3.7.
- *Other type field* – Opens a new window on top of the main window for defining the field/subfield selected (highlighted) in the sample packet. If this button is pressed, the content of the field/subfield will be regarded as other than integer type of data. The opening window is explained in Section 6.3.7.
- *Clear defined field* – Deletes the definition of the selected field/subfield and removes it from the protocol tree. A field can be selected for deletion by first highlighting it in the protocol tree frame and then clicking the highlighted area. This will result in highlighting the field with gray in both the sample packet frame and the protocol tree frame. Only one field should be selected



for deletion at a time. These selections can be removed by clicking either of the frames with the secondary mouse button (right click). If the field to be deleted has subfields, they will be deleted too. In this case the user will be warned and asked to confirm the action.

- *Create dissector* – Opens a new window on top of the main window for defining the last necessary information before a Lua dissector can be generated for the target protocol. The opening window is explained in Section 6.3.8.

#### 6.3.4. Help window

In Figure 14, the help window for displaying the usage instructions of Lua Dissector Generator is illustrated. The scrollable text frame contains the manual in a clear and structured form. There are only two buttons in the window: *Find* and *Close*. The first one opens a small search window with which the user can search for words and phrases in the manual. The word/phrase is entered to the text field and after clicking *OK* the searched word/phrase will be highlighted in the manual and the text frame is scrolled automatically to the right position. The second button simply closes the help window.

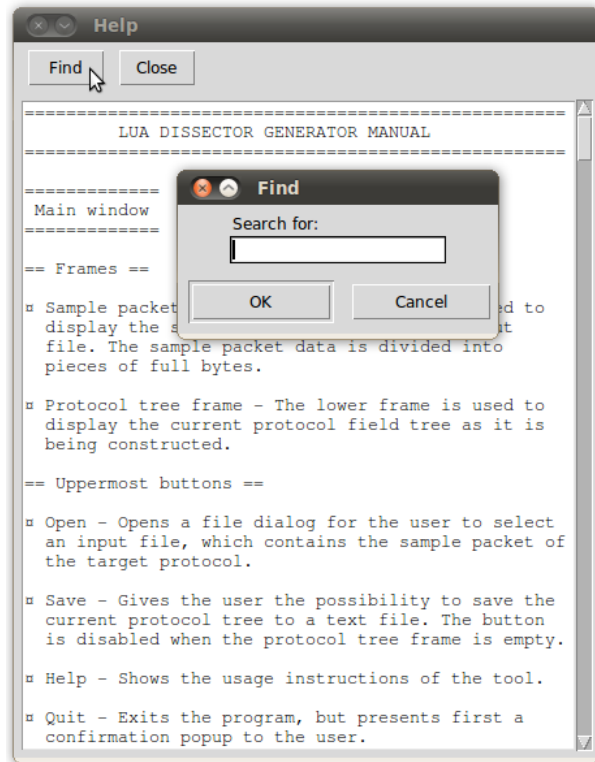


Figure 14. Help window.

### 6.3.5. Define protocol information window

A screenshot of the window with example inputs is presented in Figure 15. The buttons at the bottom of the window are quite self-explanatory: *Apply* means applying the defined information to the protocol and *Cancel* means canceling the definition process. Another option is to press Enter or Esc (Escape) on the keyboard respectively. In the following, the meaning and usage of the input fields and radio buttons are explained.

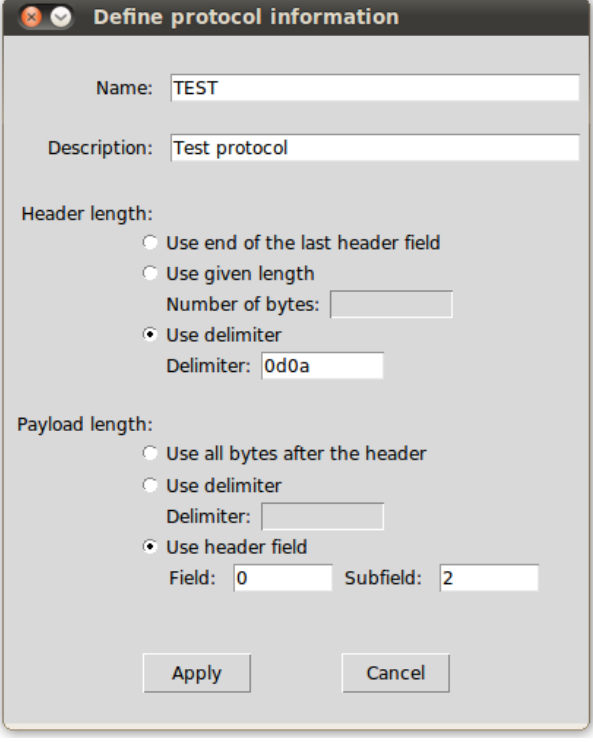
- *Name* – The user should write the name of the protocol in this text field. The name must consist of standard ASCII characters, but if it contains only space characters it is rejected.
- *Description* – The user should write a short description of the protocol in this text field. The same rule applies to the description: it must consist of standard ASCII characters, but if it contains only space characters it is rejected.

#### Header length

The first three radio buttons are used to choose the way how the length of the packet header is determined. Two of the radio buttons involve also an input field, which will be enabled when the corresponding option is selected. Only one of the three alternatives can be chosen.

- *Use end of the last header field* – The packet header ends to the last defined field, i.e., to the last byte of the last defined field.
- *Use given length* – The length of the packet header will be given manually by the user. The length should be given as the number of bytes and entered into the input field. The input must be *a*) a valid integer (e.g., “012” is not a valid integer), *b*) greater than zero and *c*) less than the number of bytes in the packet.
- *Use delimiter* – The length of the packet header will be determined by a specified delimiter. A delimiter is an array of characters the purpose of which is to act as a boundary between data. The delimiter must be given as full bytes of valid hexadecimal values, either beginning with “0x” or not (e.g., “0x0A0A” or “0A0A”). The case does not matter and spaces are ignored, for example, “1A2b” equals “1 a 2B”. Each hexadecimal value must be between 0x20 and 0x7E, or one of the three acceptable control characters: 0x09, 0x0A, or 0x0D. In addition, the given delimiter must be found once and only once inside the sample packet, except if the delimiter of the payload is the same, in which case it may appear twice.

If the option *Use given length* or *Use delimiter* is selected, the tool checks that the header fields will not exceed the specified header length. In case the user tries to define a header field that crosses the header boundaries the tool will throw an error popup with an appropriate message.



The image shows a 'Define protocol information' dialog box. It has a title bar with a close button, a maximize button, and a checkmark button. The main area contains the following fields and options:

- Name:** A text field containing 'TEST'.
- Description:** A text field containing 'Test protocol'.
- Header length:** A section with three radio buttons:
  - ☐ Use end of the last header field
  - ☐ Use given length
  - ☒ Use delimiter
 Below these is a 'Number of bytes:' text field (disabled) and a 'Delimiter:' text field containing '0d0a'.
- Payload length:** A section with three radio buttons:
  - ☐ Use all bytes after the header
  - ☐ Use delimiter
  - ☒ Use header field
 Below these is a 'Delimiter:' text field (disabled), and for the selected 'Use header field' option, there are two text fields: 'Field:' containing '0' and 'Subfield:' containing '2'.
- At the bottom are 'Apply' and 'Cancel' buttons.

Figure 15. Define protocol information window.

## Payload length

The last three radio buttons are used to choose the way how the length of the packet payload is determined. Two of the radio buttons involve also an input field, which will be enabled when the corresponding option is selected. Only one of the three alternatives can be chosen.

- *Use all bytes after the header* – All remaining bytes after the packet header are considered as the payload.
- *Use delimiter* – The length of the packet payload will be determined by a specified delimiter. The same rules apply as in the case of the header delimiter. However, there is one additional rule: the payload delimiter must be after the header.
- *Use header field* – A specific header field/subfield defines the length of the payload. The field is determined by giving its index number in the packet header and the subfield is determined by giving its index number in this particular field. Both the field and the subfield indexing start from 0. Both indices must be given as valid, non-negative integers, the field index must be smaller than the number of defined fields, and the subfield index must be smaller than the number of defined subfields inside the field, otherwise an error is thrown. The content of the header field/subfield is interpreted as an integer which indicates the payload length. If only the field index is given, the payload length is obtained from a field, but if also the subfield

index is given, the payload length is obtained from a subfield. An error is thrown if only the subfield index is given. In addition, the tool checks that there is at least as many bytes left after the header as the payload length field/subfield indicates.

### 6.3.6. Define delimiters window

The window for defining the field and subfield delimiters is shown in Figure 16. The buttons at the bottom of the window have the same functionalities as in the previous window: *Apply* means applying the specified delimiters to the packet and *Cancel* means canceling the definition process. Another option is to press Enter or Esc (Escape) on the keyboard respectively.

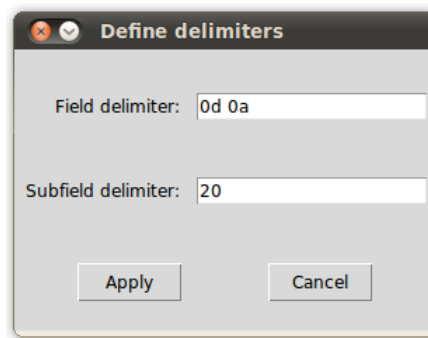


Figure 16. Define delimiters window.

There are two text fields in the window for user input: *Field delimiter* and *Subfield delimiter*. The following rules apply to using either of the input fields, but only the field delimiter is required, the subfield delimiter is optional. The delimiters must be given as full bytes of valid hexadecimal values, either beginning with “0x” or not (e.g., “0x0d0a” or “0d 0a”). The case does not matter and spaces are ignored, for example, “0D0a” equals “0 d 0A”. Each hexadecimal value must be between 0x20 and 0x7E, or one of the three acceptable control characters: 0x09, 0x0A, or 0x0D.

### 6.3.7. Define field information windows

The two different *Define field information* windows explained in the following are used for defining the protocol fields and subfields. The buttons at the bottom of the both windows have the same functionalities as in the previous windows: *Apply* means applying the defined information to the selected protocol field or subfield, and *Cancel* means canceling the definition process. Another option is to press Enter or Esc (Escape) on the keyboard respectively.

More details about the field types and other parameters can be found from the Wireshark User's Guide<sup>4</sup>. Especially the section "Functions for writing dissectors" is recommended to be viewed, but the whole chapter "Lua Support in Wireshark" is considered useful.

## Integer type field

In Figure 17, the window for defining the necessary information for an integer type field is demonstrated. The first two input fields are required. If either of them is left out, an error popup is triggered on *Apply*.

- *Type* – The type of the field. The type is selected with the drop-down list. The alternative types are: *uint8*, *uint16*, *uint24*, *uint32*, *uint64*, *int8*, *int16*, *int24*, *int32*, *int64*, and *framenum*. The *intX/uintX* types represent *X*-bit integers/unsigned integers and the *framenum* type represents a frame number.
- *Abbreviation* – The abbreviated name of the field, which is used in Wireshark filters. The abbreviation *a*) must consist of standard ASCII characters, *b*) must not contain any dots (.) and *c*) must not consist only of spaces. Spaces are allowed, but they will be replaced with underscores (\_).

Figure 17. Define integer type field window.

<sup>4</sup>[http://www.wireshark.org/docs/wsug\\_html\\_chunked/](http://www.wireshark.org/docs/wsug_html_chunked/)

The rest of the input fields are optional, but can still be very useful by offering more information about the fields during the actual dissection process in Wireshark.

- *Name* – The actual name of the field, which appears in the dissection tree in Wireshark. The name must consist of standard ASCII characters, but not only spaces.
- *Description* – The description of the field. The description must consist of standard ASCII characters, but not only spaces.
- *Base* – The base of the representation: DEC (decimal), HEX (hexadecimal), OCT (octal). The default option is Unknown, which corresponds to base NONE in Wireshark.
- *Valuestring* – A table containing the text that corresponds to the numerical key values. Elements (key-value pair) of the table must be separated by commas. Each element must be given in form:  $[N] = "desc"$ , where  $N$  denotes an integer value (key) of the field and  $desc$  the corresponding textual description of the value as a string. The string value can contain only alphabets, digits, and underscores. The key must be in square brackets, the string value in single or double quotes, and the key-value pair must be separated by an equal sign. An example input is shown in Figure 17. This table is referred in Wireshark to interpret the field value and show the corresponding description of the value in the dissection tree.
- *Bitmask* – The bitmask to be used for this field. The bitmask is used to indicate which bits of the selected bytes are actually included in the field. The mask must begin with “0x” prefix and it must be given in hexadecimal values. The mask must also be as long as the selected field in bytes. For example, if two bytes were selected/highlighted from the sample packet, the bitmask could be given as “0x0FFF” indicating that the highest nibble (half of a byte) is not a part of the field.

### Other type field

In Figure 18, the window for defining the necessary information for an other (than integer) type field is demonstrated. As in the previous case, the first two input fields are required. If either of them is left out, an error popup is triggered on *Apply*.

- *Type* – The type of the field. The type is selected with the drop-down list. The alternative types are: *float*, *double*, *string*, *stringz*, *bytes*, *ubytes*, *bool*, *ipv4*, *ipv6*, *ether*, *ipxnet*, *absolute\_time*, *relative\_time*, *pcre*, *oid*, *guid*, and *ewi64*. The Wireshark User’s Guide and documentation should be examined if explanations for these types are needed. The reason why they are not explained in here is that they are Wireshark related specifics, not LuDis related.

- *Abbreviation* – The abbreviated name of the field, which is used in Wireshark filters. The abbreviation *a)* must consist of standard ASCII characters, *b)* must not contain any dots (.) and *c)* must not consist only of spaces. Spaces are allowed, but they will be replaced with underscores (\_).

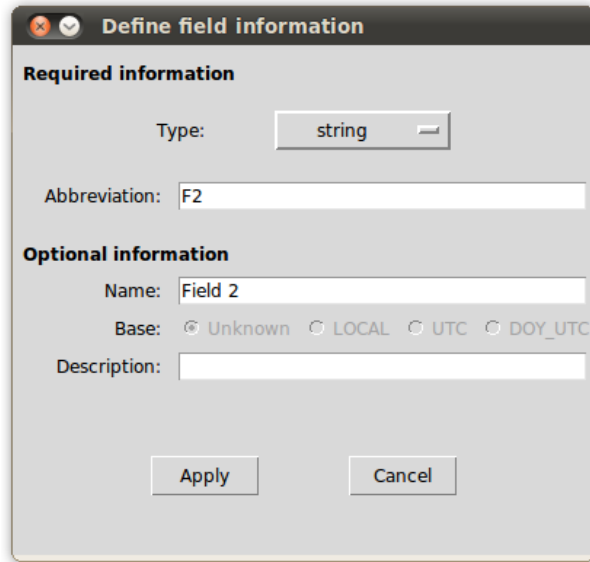


Figure 18. Define other type field window.

The rest of the input fields are optional, but can still be very useful by offering more information about the fields during the actual dissection process in Wireshark.

- *Name* – The actual name of the field, which appears in the dissection tree in Wireshark. The name must consist of standard ASCII characters, but not only spaces.
- *Base* – The base of the time representation: LOCAL, UTC (Coordinated Universal Time), DOY-UTC. The default option is Unknown, which corresponds to base NONE in Wireshark. The base options are enabled only when *absolute\_time* is selected as the field type. See the Wireshark User's Guide for more information.
- *Description* – The description of the field. The description must consist of standard ASCII characters, but not only spaces.

### 6.3.8. Create protocol dissector window

The window for finishing the process and generating a Lua dissector for the defined protocol is shown in Figure 19. The buttons at the bottom work basically the same way as *Apply* and *Cancel* in the other windows: *Create* applies the given information, creates a dissector in Lua according to the protocol definitions, and

saves it into the specified directory; *Cancel* simply cancels this phase and closes the window. Pressing Enter or Esc (Escape) on the keyboard will trigger the same actions respectively.

Only the first field is required to be filled. The button is linked to the text field next to it.

- *Save in..* – Opens a directory dialog for selecting the directory in which the generated dissector will be saved. The path of the selected directory will be shown in the text field next to the button. The filename is generated from the specified protocol name by converting the characters to lower case, replacing spaces with underscores, and adding a “.lua” suffix/file extension after the filename.

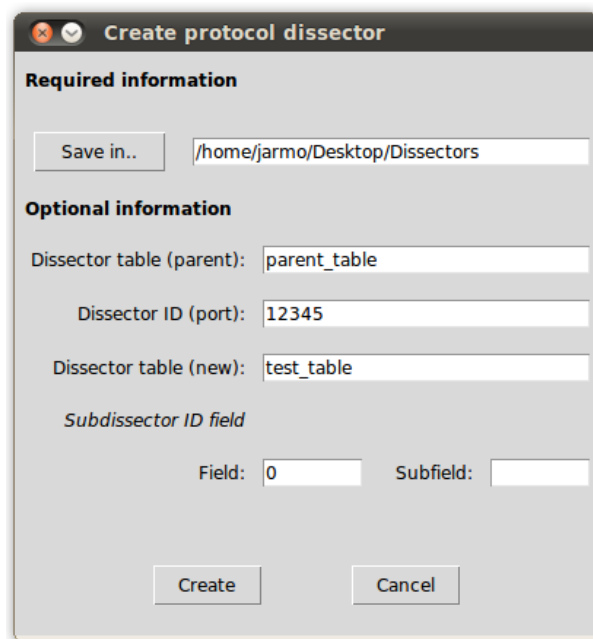


Figure 19. Create protocol dissector window.

The rest of the input fields are optional and concern the relations between different dissectors. However, if the parent dissector table is specified, the dissector ID (port number) must be given too, and vice versa.

- *Dissector table (parent)* – The short name of a table of subdissectors of a particular protocol (e.g., TCP subdissectors, such as dissectors for HTTP, SMTP, and SIP are added to dissector table “tcp.port”). This dissector table refers to the parent dissector of the dissector at hand. In other words, the newly defined protocol would be considered as a subprotocol of the protocol that owns this dissector table. Hence, the new dissector would be dissecting the payloads of its parent dissector. Wireshark dissector tables and their short names can be checked in Wireshark from menu *Internals* → *Dissector tables*. The table name must consist of standard, lowercase ASCII characters, but whitespaces are not allowed.



- *Dissector ID (port)* – The dissector identifier related to the parent dissector table, i.e., the port number that this new protocol uses. The ID must be a valid, non-negative integer and the same number must not be already assigned to another dissector of the same dissector table. However, verifying the latter one is the user's responsibility: reserved port numbers/IDs can be checked in Wireshark from menu *Internals* → *Dissector tables*.
- *Dissector table (new)* – The short name for a new table of subdissectors of this particular protocol. If this input field is not left empty, a new dissector table will be created for the protocol at hand. Hence, the dissector to be generated can act as a parent dissector for other dissectors, so that the packet payloads of this protocol can be forwarded to dissectors of subordinate protocols. The table name must consist of standard, lowercase ASCII characters, but whitespaces are not allowed. The name must also be unique, meaning that there must not be already a dissector table with the same name in Wireshark.
- *Subdissector ID field* – These two input fields specify the field/subfield of a packet of this protocol that will be used to indicate the subdissector to which the payload part of the packet will be given. In other words, the contents of the specified field/subfield will be interpreted as an integer which determines the ID (port number) of the subdissector. This ID is searched from the dissector table of the protocol and if a match is found, the payload part will be passed on to the corresponding subdissector.
  - *Field*: The field number (index) of the subdissector ID field. The number must be a valid, non-negative integer and smaller than the number of fields in the packet. In other words, the number must refer to an existing and defined field. The field indexing starts from 0. The field must also be inside the packet header. If only the field number is given, the subdissector ID is obtained from a header field.
  - *Subfield*: The subfield number (index) of the subdissector ID field. The number must be a valid, non-negative integer and smaller than the number of subfields in the specified field. In other words, the number must refer to an existing and defined subfield. The subfield indexing starts also from 0. If both the field and the subfield number are given, the subdissector ID is obtained from a header subfield. An error is thrown if only the subfield index is given.

### 6.3.9. *Popup windows*

The tool makes use of many different popup windows. The popups are utilized throughout the dissector generation process to guide and inform the user. In Figure 20, some examples of possible popup windows are demonstrated. Subfigure (a) depicts an information popup, subfigure (b) a popup prompting for confirmation, and subfigures (c) and (d) popups declaring an error in the program execution.

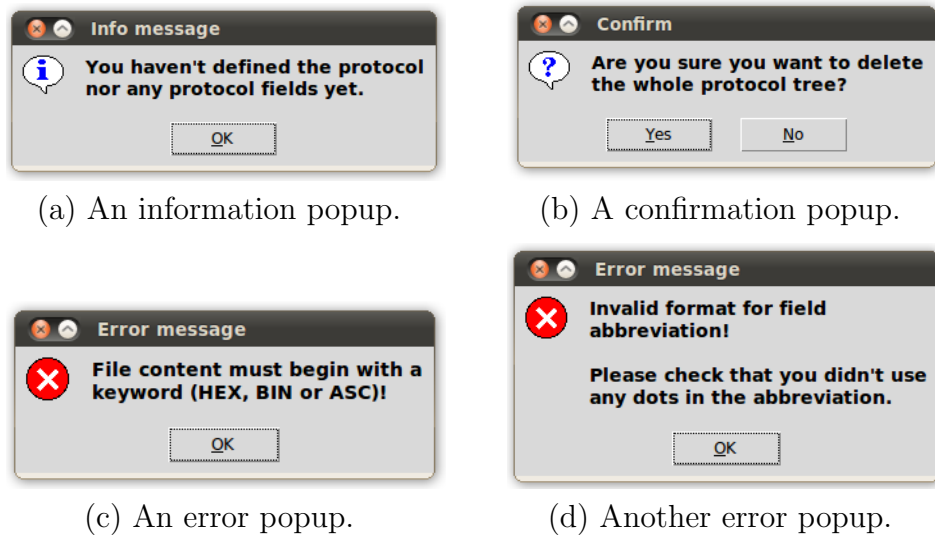


Figure 20. Examples of popups.

#### 6.3.10. How to use the generated Lua dissectors in Wireshark?

The following steps are needed to use the generated Lua dissectors in Wireshark.

1. Download and install the latest version of Wireshark from <http://www.wireshark.org/download.html>.
2. Check that Lua support is enabled in Wireshark. In the latest version it should be enabled by default. It can be checked whether the installed version of Wireshark has Lua support enabled by going in Wireshark to *Help* → *About Wireshark* and looking if Lua version is mentioned in the “Compiled with...” paragraph. If no mention of Lua can be found, the support can be enabled by modifying a line in file *init.lua* located in the global configuration directory of Wireshark (the path of this directory is similar to the one in Figure 21). To enable Lua the line variable *disable\_lua* must be set to *false*.
3. Locate the *Personal configuration* and the *Personal Plugins* directories. The paths of these directories can be checked by opening Wireshark, navigating to *Help* → *About Wireshark* and clicking the *Folders* tab. These actions will lead to a window similar to the window presented in Figure 21.
4. Copy the generated Lua dissector file
  - (a) either into the *Personal configuration* directory if the dissector creates a new dissector table, i.e., if the dissector acts as a parent dissector for other dissectors. Next, open the file *init.lua* residing in this directory and assuming that the name of the dissector file is *parent\_dissector.lua*, insert the following line of code: `dofile("parent_dissector.lua")`.

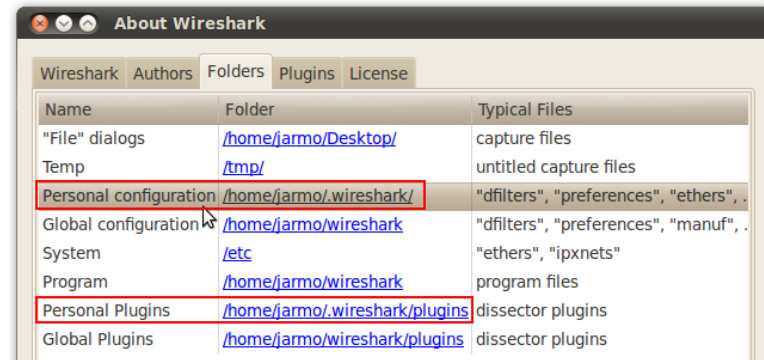


Figure 21. Wireshark folders.

This ensures that the *parent\_dissector.lua* is loaded before the dissector plugins (subdissectors) located in the *Global Plugins* and *Personal Plugins* directories.

- (b) or into the *Personal Plugins* directory if the dissector is connected to an existing dissector table, i.e., if the dissector acts as a subdissector.
- 5. Restart Wireshark. To make sure that the new script is loaded, navigate to *Help* → *About Wireshark* and choose the *Plugins* tab. The added dissector should appear in the list as a plugin of type "lua script".
- 6. Open a capture file that contains packets of the specified protocol or start a live capture.

The command line version of Wireshark, TShark<sup>5</sup>, can also be used to load Lua scripts. This can be done by going to the Wireshark installation directory and entering the following command in the terminal:

`tshark -X lua_script:<path_to_file>` (without the "<" and ">" signs).

The Wireshark User's Guide is recommended for more information on using Lua dissectors in Wireshark. Especially the chapter "Lua Support in Wireshark" might be useful.

<sup>5</sup><http://www.wireshark.org/docs/man-pages/tshark.html>

## 7. EVALUATION OF LUA DISSECTOR GENERATOR

In this chapter, the functionality of Lua Dissector Generator is evaluated with a couple of test cases. The cases demonstrate the dissector generation process all the way from the definition of the protocol and its fields to generating a protocol dissector and using it in Wireshark. The test cases are introduced with such accuracy that the results are reproducible.

### 7.1. Test case 1

Test case 1 demonstrates the dissector generation process for a custom protocol that operates on top of the UDP protocol. Before having a specific dissector for this protocol Wireshark is incapable of structuring and interpreting the packets of the protocol. The result of dissecting a packet of the custom protocol in Wireshark without an appropriate dissector is shown in Figure 22. As can be seen, the relevant bytes (the payload part of the UDP packet) are simply regarded as a chunk of meaningless data.

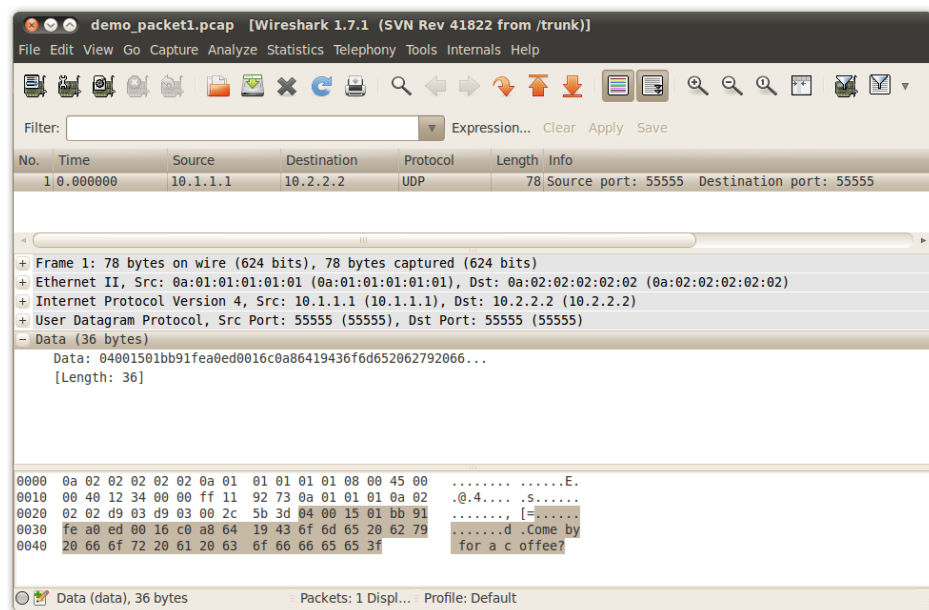


Figure 22. Wireshark dissection result before having an appropriate dissector.

### Protocol definition

In Table 4, the information used for the protocol definition is presented. The first row specifies the type of the information and the second row the given information. The protocol information is defined as described in Section 6.3.5.

Table 4. Protocol definition of test case 1

<i>Name</i>	<i>Description</i>	<i>Header length</i>	<i>Payload length</i>
DEMO1	DEMO1 Test Protocol	Use given length: 15	Use header field: F: 1 S: –

### Field definitions

In Table 5, the information used for the definitions of the protocol fields and subfields is presented. The first row specifies the type of the information and the subsequent rows the given information for each field and subfield correspondingly.

Table 5. Field definitions of test case 1

<i>Bytes</i>	<i>Type</i>	<i>Abbreviation</i>	<i>Name</i>	...
0	uint8	f0	Message ID	...
1-2	uint16	f1	Payload length	...
3-8	bytes	f2	Source info	...
3-4	uint16	sf2-0	Source port	...
5-8	ipv4	sf2-1	Source IP	...
9-14	bytes	f3	Destination info	...
9-10	uint16	sf3-0	Destination port	...
11-14	ipv4	sf3-1	Destination IP	...

...	<i>Description</i>	<i>Base</i>	<i>Valuestring</i>	<i>Bitmask</i>
...	—	—	—	—
...	—	DEC	—	—
...	Information about the sender	—	—	—
...	—	DEC	[22]="SSH", [443]="HTTPS"	—
...	—	—	—	—
...	Information about the receiver	—	—	—
...	—	DEC	[22]="SSH", [443]="HTTPS"	—
...	—	—	—	—

The header fields and subfields inside the packets of the protocol have fixed lengths. Therefore the bytes of the fields and subfields are selected by highlighting them in the sample packet with mouse. The process of defining protocol fields is described in Section 6.3.7.

### Dissector details

In Table 6, the necessary dissector related details are presented. The names of the existing dissector tables can be checked from Wireshark by navigating to *Internals* → *Dissector tables* and searching for the corresponding short name of the desired dissector table.

Table 6. Dissector details of test case 1

<i>Dissector table (parent)</i>	<i>Dissector ID (port)</i>	<i>Dissector table (new)</i>	<i>Subdissector ID field</i>
udp.port	55555	—	F: — S: —

### Results

After copying the generated dissector file into the *Personal Plugins* directory and restarting Wireshark, the new dissector is utilized and Wireshark is capable of understanding packets of the custom protocol. In Figure 23, the dissection result of the previously unidentified packet is depicted.

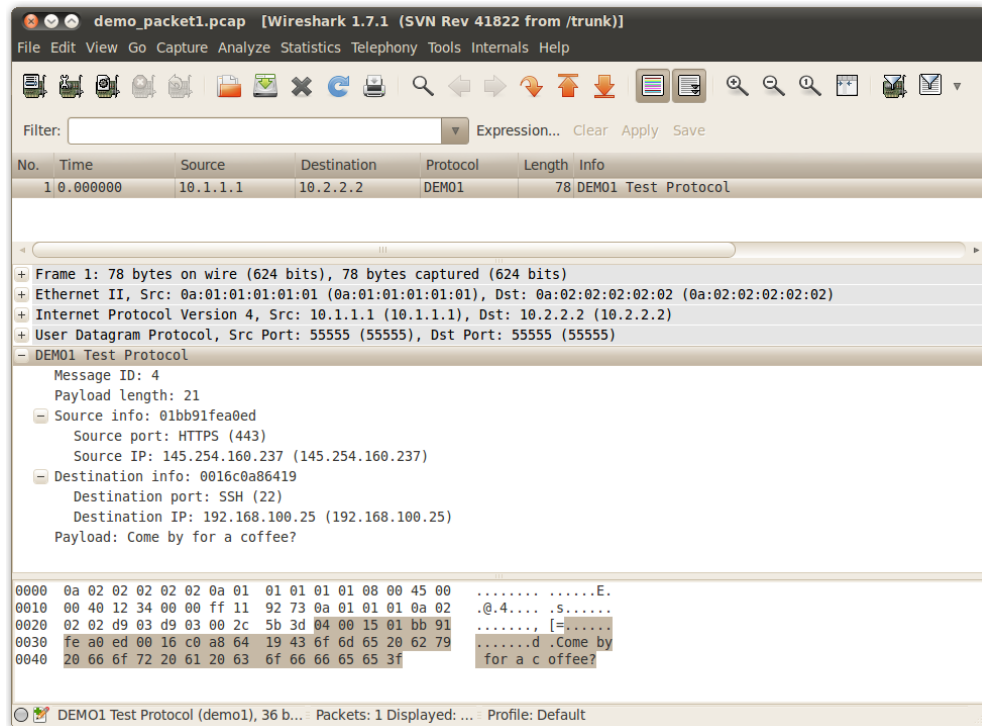


Figure 23. Wireshark dissection result when using the generated dissector.

The header fields and subfields are dissected according to the definitions, the header and the payload are separated properly, the IP addresses are eligible, and the valustrings are referenced correctly by the port numbers. Compared to Figure 22, it is evident that the protocol packet is much more understandable and informative after generating a specific dissector for the custom protocol. The Lua code of the generated protocol dissector is presented in Appendix 1.

## 7.2. Test case 2

Test case 2 demonstrates the creation and usage of two related dissectors: one for a parent protocol that operates on top of the TCP protocol and one for a subprotocol that operates on top of the parent protocol. Before generating either of them, the result of Wireshark dissection is similar to the one in Figure 22: the protocol packet (the payload part of the TCP packet) appears to be just random bytes.

### 7.2.1. Generating the parent dissector

The task of the parent dissector is to dissect the first part of the TCP payload. It parses and interprets the parent protocol header fields in the packet and again the payload of the parent protocol packet is passed on to the subdissector.

#### Protocol definition

In Table 7, the information used for the definition of the parent protocol is presented. The used field and subfield delimiter are in turn named in Table 8.

Table 7. Parent protocol definition of test case 2

<i>Name</i>	<i>Description</i>	<i>Header length</i>	<i>Payload length</i>
DEMO2	DEMO2 Parent Protocol	Use end of the last header field	Use delimiter: 0d0a

Table 8. Field and subfield delimiter of the parent protocol

<i>Field delimiter</i>	<i>Subfield delimiter</i>
3a	2f

## Field definitions

In Table 9, the information used for the definitions of the parent protocol fields and subfields is presented. Since the header fields and subfields inside the packets are separated by delimiters, they can be of variable lengths. The fields are defined according to the given indices by selecting the corresponding fields/subfields with the arrow buttons in the main window of LuDis. More details about the delimiter related buttons can be found in Section 6.3.3.

Table 9. Field definitions of the parent protocol of test case 2

<i>Field index</i>	<i>Type</i>	<i>Abbreviation</i>	<i>Name</i>	...
0	uint16	f0	Port	...
1	string	f1	Server	...
2	string	f2	User	...
2-0	string	sf2-0	Full name	...
2-1	string	sf2-1	Nickname	...
2-2	uint8	sf2-2	Status	...

...	<i>Description</i>	<i>Base</i>	<i>Value</i> <i>string</i>	<i>Bitmask</i>
...	—	DEC	[6679]=“IRC SSL”, [5298]=“XMPP”	0xFFFF00
...	—	—	—	—
...	—	—	—	—
...	—	—	—	—
...	IRC nickname	—	—	—
...	—	DEC	[0]=“away”, [1]=“online”, [2]=“busy”	—

## Dissector details

In Table 10, the necessary dissector related details are presented. The dissector table named in the third column is not an existing one, but instead a new dissector table will be created with the given name. After utilizing this parent dissector in Wireshark the new dissector table can be referenced by other dissectors. This makes the chaining of dissectors possible so that dissectors can form stacks of subdissectors, in the same way as network protocols form stacks. Letters F and S in the fourth column denote *Field* and *Subfield*, respectively. In this case, the first field is responsible for addressing the subdissector to which the payload of the parent protocol is dispatched.



Table 10. Dissector details of the parent protocol of test case 2

<i>Dissector table (parent)</i>	<i>Dissector ID (port)</i>	<i>Dissector table (new)</i>	<i>Subdissector ID field</i>
tcp.port	9876	demo2.port	F: 0 S: –

## Results

The Lua code generated by LuDis for the parent protocol is presented in Appendix 2. The generated parent dissector file must be copied into Wireshark's *Personal configuration* directory and the file named *init.lua* must be modified according to the instructions in Section 6.3.10. After restarting Wireshark, the new dissector is utilized and Wireshark is capable of understanding packets of the parent protocol. In Figure 24, the dissection result is shown. Since there is not yet a subdissector for the payload part, it is not analyzed in any way. At this point, only the header fields of the parent protocol are interpreted. Now the packet contents are much clearer: the first field denotes a network port, the second field an IRC (Internet Relay Chat) server, and the third field along with its subfields information about an IRC user.

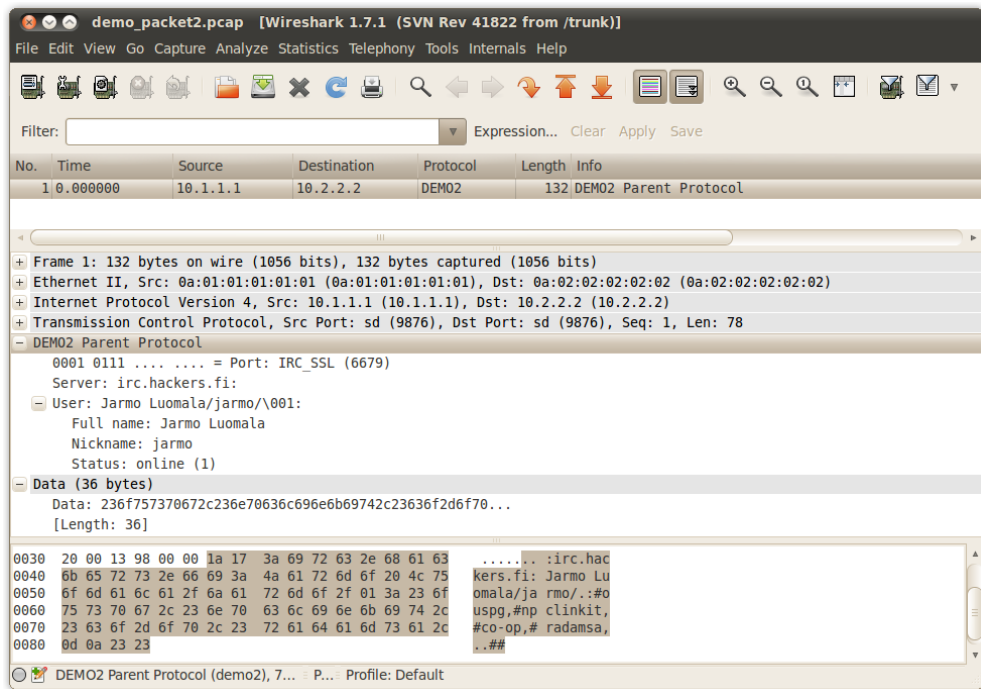


Figure 24. Wireshark dissection result when using the generated dissector.

### 7.2.2. Generating the subdissector

The task of the subdissector is to dissect and analyze the payload part of the parent protocol packet. The dissection result of the subprotocol will appear below the parent protocol in the Wireshark dissection tree.

#### Protocol definition

In Table 11, the information used for the definition of the subprotocol is presented. The used field delimiter is mentioned in Table 12.

Table 11. Subprotocol definition of test case 2

<i>Name</i>	<i>Description</i>	<i>Header length</i>	<i>Payload length</i>
DEMO3	DEMO3 Subprotocol	Use delimiter: 0d0a	Use all bytes after the header

Table 12. Field and subfield delimiter of the subprotocol

<i>Field delimiter</i>	<i>Subfield delimiter</i>
2c	—

#### Field definitions

In Table 13, the information used for the definitions of the subprotocol fields is presented. Again the header fields are separated by delimiters and thus they can be of variable lengths. The fields are selected and defined in the same way as in the case of the parent protocol.

Table 13. Field definitions of the subprotocol of test case 2

<i>Field index</i>	<i>Type</i>	<i>Abbreviation</i>	<i>Name</i>	...
0	string	ch1	Channel 1	...
1	string	ch2	Channel 2	...
2	string	ch3	Channel 3	...
3	string	ch4	Channel 4	...

...	<i>Description</i>	<i>Base</i>	<i>Valuestring</i>	<i>Bitmask</i>
...	—	—	—	—
...	—	—	—	—
...	—	—	—	—
...	—	—	—	—

## Dissector details

In Table 14, the necessary dissector related details are presented. Now the first column contains the short name of the dissector table that is created by the parent dissector. The port number given in the second column corresponds to the contents of the subdissector ID field of the parent protocol. See the value of the first header field (Port) of DEMO2 Parent Protocol in Figure 24. This is how dissectors can be connected with each other in order to form dissection chains, in which a subsequent dissector acts as a subdissector for the previous one.

Table 14. Dissector details of the subprotocol of test case 2

<i>Dissector table (parent)</i>	<i>Dissector ID (port)</i>	<i>Dissector table (new)</i>	<i>Subdissector ID field</i>
demo2.port	6679	—	F: — S: —

## Results

The Lua code generated by LuDis for the subprotocol is presented in Appendix 3. This time the dissector file must be copied again into Wireshark's *Personal Plugins* directory, because the dissector does not act as a parent for any other dissector. After restart Wireshark is able to understand and dissect both the parent protocol and the subprotocol part of the packet.

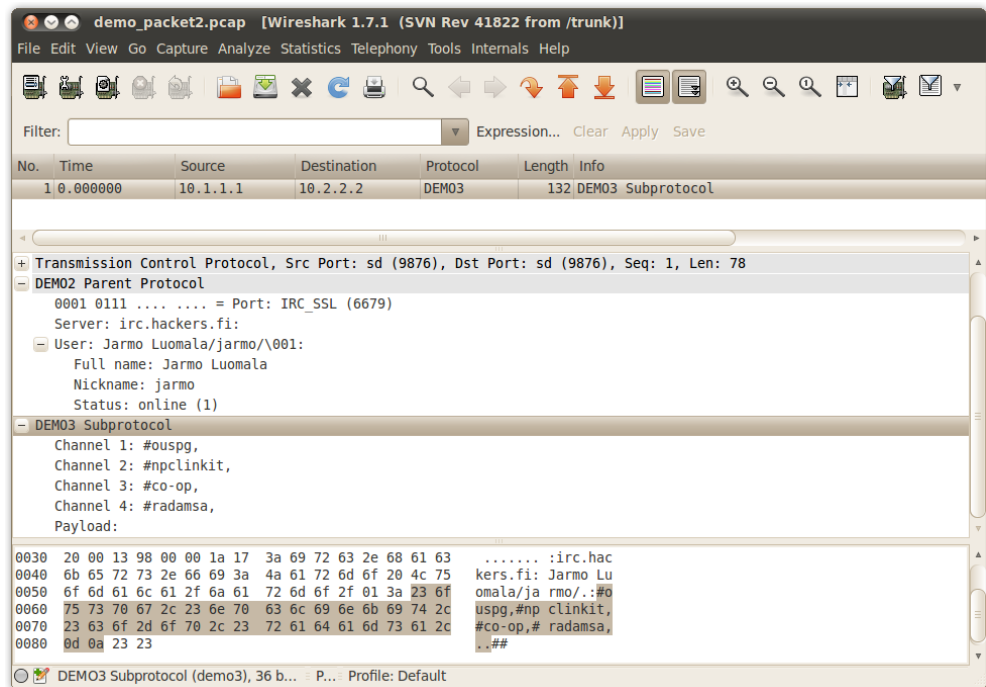


Figure 25. Wireshark dissection result when using both generated dissectors.

It is now revealed that the subprotocol part presents a list of IRC channels. In this part, there is no payload at all. The last two bytes (two # signs) of the whole network packet are not included in the DEMO3 Subprotocol packet. This is because it is the payload of the parent protocol and the payload delimiter of the parent protocol (0d0a in hexadecimals) is found before the last two bytes.

### 7.3. Discussion

In this chapter, the performance of LuDis was demonstrated and evaluated. The generated dissectors were proved to work appropriately in Wireshark and the test packets were dissected according to the specified rules. In addition, all other dissector generators found on the Internet require knowledge of some programming language or formal language, but this is not the case for LuDis. For this reason, LuDis stands out from the other dissector generators.

During the development of the tool it was noted that there are too many different rules and features among protocols to have time to implement all possible options and combinations. Decisions had to be made to exclude some options that might be necessary when specifying certain protocols. For example, the tool lacks the possibility to define trailer fields (supplemental data after the payload) and to determine the header length by using a header field. It should also be possible to interpret the content of the header field which indicates the payload length as something else than an integer, such as a coefficient for some particular constant value. These are examples of features that could be added to the tool in the future.

As an afterthought, it is worth mentioning that it would have been a better choice to use PyQt<sup>1</sup> for developing the GUI instead of Tkinter. PyQt is Python bindings for the Qt framework<sup>2</sup>, which provides, for instance, a handy drag-and-drop feature for easier GUI development. Tkinter turned out to be quite cumbersome to use and a simpler way to place GUI widgets would have saved a lot of time and effort. Use of the Qt framework would have also encouraged the use of the model-view-controller pattern, which would have made the software more modular and future updates easier.

Despite the several challenges during the implementation process, the tool reached a very satisfying level of performance. The user interface is simple enough to use and provides necessary feedback in error situations. Lua Dissector Generator removes the burden of coding protocol dissectors manually and helps to examine custom protocol packets and to understand the packet level better.

---

<sup>1</sup><https://wiki.python.org/moin/PyQt>

<sup>2</sup><http://qt.digia.com/>

## 8. CONCLUSION

In this thesis, packet analyzers and protocol dissectors were studied. First, some basics of networks and their layered nature were introduced in order to help realize the purpose and power of packet analyzers. Next, the functionalities and some typical uses of packet analyzers were mentioned and a few widely used tools were listed. Information security aspects were discussed by demonstrating the usage of packet analyzers not only in some common attacks compromising security but also in means to protect it. To conclude the background theory part and lead to the empirical part of the thesis, Wireshark dissectors were explained and a couple of different dissector generators were presented.

A tool for generating protocol dissectors for Wireshark was implemented and evaluated as a part of this thesis. The tool, Lua Dissector Generator (LuDis), has a graphical user interface, which makes specifying the protocol rules easier. The usage of the tool was explained in detail all the way from launching the software to utilizing the generated dissectors in Wireshark. Both fixed length and variable length protocol fields and packets can be defined. Dissectors can be generated to act merely as subdissectors or also as parents for other dissectors. The tool itself was implemented in Python, but the dissectors are generated in Lua programming language, because of its simplicity and understandability compared to C.

The objectives of the thesis were achieved. The developed software and the executed test cases prove that it is possible to create a tool that automates protocol dissector generation for Wireshark without requiring programming skills from the user. Lua Dissector Generator makes the creating of protocol dissectors easier by removing the need to know the specifics of how to code a dissector. In fact, it can be used without having programming skills at all. Only knowledge of the protocol at hand and some experience with Wireshark are needed.

The dissection results presented in the tool evaluation part show that the generated dissectors work very well. Wireshark provides a much more understandable and informative outcome for custom protocols when it is extended with appropriate self-made dissectors. New protocols are constantly developed and consequently new dissectors are needed at the same rate. With the help of packet analyzers and suitable dissectors, the information security of networks and their users can be significantly improved. This is because the better we expose and understand the packet level, the better we can control our networks, detect malicious activities, and solve occurring problems.

The implemented tool is not perfect, but the results are promising enough to encourage researching the subject more and developing the tool further. With more features, more comprehensive options for specifying the protocol rules, and more thorough testing, Lua Dissector Generator could be an excellent aid in network monitoring and troubleshooting.

## 9. REFERENCES

- [1] Sanders C. (2007) Practical Packet Analysis: Using Wireshark to Solve Real-World Network Problems. No Starch Press, Incorporated, San Francisco, CA, USA, 194 p.
- [2] Orebaugh A.D. & Ramirez G. (2004) Ethereal Packet Sniffing. Syngress Publishing, 468 p.
- [3] de Vivo M., de Vivo G.O. & Isern G. (1998) Internet security attacks at the basic levels. ACM SIGOPS Operating Systems Review 32, pp. 4–15.
- [4] Nayak G.N. & Samaddar S.G. (2010) Different flavours of Man-In-The-Middle attack, consequences and feasible solutions. In: 3rd IEEE International Conference on Computer Science and Information Technology – ICCSIT '10, vol. 5, pp. 491–495.
- [5] Wireshark: network protocol analyzer (accessed 6.2.2013). URL: <http://www.wireshark.org/>.
- [6] IEEE, IEEE Xplore digital library (accessed 6.2.2013). URL: <http://ieeexplore.ieee.org/>.
- [7] Tanenbaum A.S. (2003) Computer Networks. Prentice Hall, 4th ed., 384 p.
- [8] Cerf V.G. & Kahn R.E. (1974) A protocol for packet network intercommunication. IEEE Transactions on Communications 22, pp. 637–648.
- [9] Ansari S., Rajeev S. & Chandrashekar H. (2002-2003) Packet sniffing: a brief introduction. Potentials, IEEE 21, pp. 17–19.
- [10] Qadeer M.A., Khan A.H., Habeeb A.A. & Hafeez M.A. (2010) Bottleneck analysis and traffic congestion avoidance. In: Proc. of the International Conference and Workshop on Emerging Trends in Technology – ICWET '10, ACM, New York, USA, pp. 273–278.
- [11] Fuentes F. & Kar D.C. (2005) Ethereal vs. Tcpdump: a comparative study on packet sniffing tools for educational purpose. Journal of Computing Sciences in Colleges 20, pp. 169–176.
- [12] Chomsiri T. (2008) Sniffing packets on LAN without ARP spoofing. In: 3rd International Conference on Convergence and Hybrid Information Technology – ICCIT '08, vol. 2, IEEE Computer Society, pp. 472–477.
- [13] Chomsiri T. (2007) HTTPS hacking protection. In: 21st International Conference on Advanced Information Networking and Applications Workshops – AINAW '07, vol. 1, pp. 590–594.
- [14] SecTools.Org: Top 125 network security tools (accessed 6.2.2013). URL: <http://sectools.org/tag/sniffers/>.

- [15] Top 10 data/packet sniffing and analyzer tools for hackers (accessed 6.2.2013). URL: <http://www.internetgeeks.org/tech/hacking/top-10-data-packet-sniffing-analyzer-tools-hackers/>.
- [16] Cain & Abel – password recovery tool for Microsoft Operating Systems (accessed 6.2.2013). URL: <http://www.oxid.it/cain.html>.
- [17] tcpdump – Powerful command-line packet analyzer (accessed 6.2.2013). URL: <http://www.tcpdump.org/>.
- [18] Ettercap – Comprehensive suite for mitm attacks (accessed 6.2.2013). URL: <http://ettercap.github.com/ettercap/>.
- [19] Clarified Analyzer – Collaborative analysis and visualization of complex networks (accessed 6.2.2013). URL: <https://www.clarifiednetworks.com/Clarified%20Analyzer>.
- [20] Kenttälä J., Viide J., Ojala T., Pietikäinen P., Hiltunen M., Huhta J., Kenttälä M., Salmi O. & Hakanen T. (2009) Clarified recorder and analyzer for visual drill down network analysis. In: *Passive and Active Network Measurement, Lecture Notes in Computer Science*, vol. 5448, Springer Berlin / Heidelberg, pp. 122–125.
- [21] Kismet – Wireless network detector, sniffer, and intrusion detection system (accessed 6.2.2013). URL: <http://www.kismetwireless.net/>.
- [22] NetStumbler – Wireless network detector and sniffer (accessed 6.2.2013). URL: <http://www.netstumbler.com/>.
- [23] Capsa – Real-time portable network analyzer (accessed 6.2.2013). URL: <http://www.colasoft.com/capsa/>.
- [24] Nmap – Free security scanner for network exploration & hacking (accessed 6.2.2013). URL: <http://nmap.org/>.
- [25] dsniff – Collection of tools for network auditing and penetration testing (accessed 6.2.2013). URL: <http://monkey.org/~dugsong/dsniff/>.
- [26] Metasploit – Penetration testing software (accessed 6.2.2013). URL: <http://www.metasploit.com/>.
- [27] The Office of the Law Revision Counsel of the U.S. House of Representatives (2012), *The United States Code*, 44 USC Sec. 3542.
- [28] Whitman M.E. & Mattord H.J. (2005) *Principles of Information Security*. Thomson Course Technology, 2nd ed., 576 p.
- [29] Álvarez G. & Petrović S. (2003) A new taxonomy of Web attacks suitable for efficient encoding. *Computers & Security* 22, pp. 435–449.
- [30] The MITRE Corporation (2011), 2011 CWE/SANS Top 25 Most Dangerous Software Errors. URL: <http://cwe.mitre.org/top25/>.

- [31] Callegati F., Cerroni W. & Ramilli M. (2009) Man-in-the-Middle attack to the HTTPS protocol. *Security Privacy, IEEE* 7, pp. 78–81.
- [32] Network Working Group, IETF (2008), The Transport Layer Security (TLS) protocol, version 1.2, IETF RFC 5246. URL: <http://tools.ietf.org/html/rfc5246>.
- [33] Network Working Group, IETF (2000), HTTP over TLS, IETF RFC 2818. URL: <http://tools.ietf.org/html/rfc2818>.
- [34] Xi Y., Xiaochen C. & Fangqin X. (2011) Recovering and protecting against dns cache poisoning attacks. In: *International Conference on Information Technology, Computer Engineering and Management Sciences – ICM '11*, vol. 2, pp. 120–123.
- [35] Sobeslav V. (2011) Computer networking and sociotechnical threats. In: *Proc. of the International Conference on Applied, Numerical and Computational Mathematics – ICANCM '11*, and *Proc. of the International Conference on Computers, Digital Communications and Computing – ICDCC '11*, World Scientific and Engineering Academy and Society (WSEAS), Stevens Point, Wisconsin, USA, pp. 75–79.
- [36] Kristol D.M. (2001) HTTP cookies: Standards, privacy, and politics. *ACM Transactions on Internet Technology* 1, pp. 151–198.
- [37] Noiumkar P. & Chomsiri T. (2008) Top 10 free web-mail security test using session hijacking. In: *3rd International Conference on Convergence and Hybrid Information Technology – ICCIT '08*, vol. 2, pp. 486–490.
- [38] Syverson P. (1994) A taxonomy of replay attacks [cryptographic protocols]. In: *Proc. of the Computer Security Foundations Workshop VII – CSFW '94*, pp. 187–191.
- [39] Kemmerer R. & Vigna G. (2002) Intrusion detection: a brief history and overview. *Computer* 35, pp. 27–30.
- [40] Qadeer M., Zahid M., Iqbal A. & Siddiqui M. (2010) Network traffic analysis and intrusion detection using packet sniffer. In: *2nd International Conference on Communication Software and Networks – ICCSN '10*, pp. 313–317.
- [41] Trabelsi Z. & Shuaib K. (2006) NIS04-4: Man in the middle intrusion detection. In: *IEEE Global Telecommunications Conference – GLOBECOM '06*, pp. 1–6.
- [42] Hart M., Manadhata P. & Johnson R. (2011) Text classification for data loss prevention. In: *Proc. of the 11th International Conference on Privacy Enhancing Technologies – PETS '11*, Springer-Verlag, Berlin, Heidelberg, pp. 18–37.
- [43] Kernighan B.W. & Ritchie D.M. (1988) *The C Programming Language*. Prentice Hall, 2nd ed., 272 p.



- [44] Wireshark documentation: README.plugins (accessed 6.2.2013). URL: <http://anonsvn.wireshark.org/wireshark/trunk/doc/README.plugins>.
- [45] Ierusalimschy R., de Figueiredo L.H. & Celes W. (2006) Lua 5.1 Reference Manual. Lua.org, 112 p.
- [46] Lua programming language (accessed 6.2.2013). URL: <http://www.lua.org/>.
- [47] DeLoura M. (2009), The Engine Survey: General results (accessed 6.2.2013). URL: <http://www.satori.org/2009/03/the-engine-survey-general-results/>.
- [48] Bergersen E., Fibichr J., Mannsverk S.J., Snarby T., Thomassen E.W., Tønder L.S. & Wien S. (2011), Automated generation of protocol dissectors for Wireshark, 299 p.
- [49] Wårre T. (2010) Creating an interpreted dissector for Wireshark. Master's thesis, Chalmers University of Technology, Göteborg, Sweden, 53 p.
- [50] van Rossum G. & Drake Jr. F.L. (2003) An Introduction to Python. A Python manual, Network Theory Limited, 120 p.
- [51] Wireshark dissectors in python (accessed 10.10.2012). URL: <http://wiki.wireshark.org/Python>.
- [52] Lutz M. (2001) Programming Python. O'Reilly & Associates, Inc., 2nd ed., 1255 p.

## 10. APPENDICES

- Appendix 1    Lua code of the Wireshark dissector for DEMO1 protocol
- Appendix 2    Lua code of the Wireshark dissector for DEMO2 protocol
- Appendix 3    Lua code of the Wireshark dissector for DEMO3 protocol

## Appendix 1. Lua code of the Wireshark dissector for DEMO1 protocol

```

-- Dissector for DEMO1 --
local proto = Proto("DEMO1", "DEMO1 Test Protocol")

-- Protocol field definitions --
local fields = proto.fields
fields.f1 = ProtoField.uint8("proto.f0", "Message ID", nil, nil, nil, nil)
fields.f2 = ProtoField.uint16("proto.f1", "Payload length", base_DEC, nil, nil, nil)
fields.f3 = ProtoField.bytes("proto.f2", "Source info", nil, nil, nil,
    "Information about the sender")
local f3_s1_valstr = { [443] = "HTTPS", [22] = "SSH", }
fields.f3_s1 = ProtoField.uint16("proto.sf2-0", "Source port", base_DEC,
    f3_s1_valstr, nil, nil)
fields.f3_s2 = ProtoField.ipv4("proto.sf2-1", "Source IP", nil, nil, nil, nil)
fields.f4 = ProtoField.bytes("proto.f3", "Destination info", nil, nil, nil,
    "Information about the receiver")
local f4_s1_valstr = { [443] = "HTTPS", [22] = "SSH", }
fields.f4_s1 = ProtoField.uint16("proto.sf3-0", "Destination port", base_DEC,
    f4_s1_valstr, nil, nil)
fields.f4_s2 = ProtoField.ipv4("proto.sf3-1", "Destination IP", nil, nil, nil, nil)
fields.payload = ProtoField.string("proto.payload", "Payload")

-- Dissector function --
function proto.dissector(buffer, pinfo, tree)

    local subtree = tree:add(proto, buffer())
    pinfo.cols.protocol = proto.name
    pinfo.cols.info = proto.description

    local packet = buffer():string()
    subtree:add(fields.f1, buffer(0,1))
    subtree:add(fields.f2, buffer(1,2))
    local subsubtree_f3 = subtree:add(fields.f3, buffer(3,6))
    subsubtree_f3:add(fields.f3_s1, buffer(3,2))
    subsubtree_f3:add(fields.f3_s2, buffer(5,4))
    local subsubtree_f4 = subtree:add(fields.f4, buffer(9,6))
    subsubtree_f4:add(fields.f4_s1, buffer(9,2))
    subsubtree_f4:add(fields.f4_s2, buffer(11,4))

    -- Handle the payload --
    local payload_length = buffer(1,2):uint()
    subtree:add(fields.payload, buffer(15, payload_length))

end

-- Dissector registration --
local parent_dt = DissectorTable.get("udp.port")
parent_dt:add(55555, proto)

```

## Appendix 2. Lua code of the Wireshark dissector for DEMO2 protocol

```

-- Dissector for DEMO2 --
local proto = Proto("DEMO2", "DEMO2 Parent Protocol")
local proto_dt = DissectorTable.new("demo2.port", "DEMO2")

-- Protocol field definitions --
local fields = proto.fields
local f1_valstr = { [5298] = "XMPP", [6679] = "IRC_SSL", }
fields.f1 = ProtoField.uint16("proto.f0", "Port", base_DEC, f1_valstr, 0xFFFF00, nil)
fields.f2 = ProtoField.string("proto.f1", "Server", nil, nil, nil, nil)
fields.f3 = ProtoField.string("proto.f2", "User", nil, nil, nil, nil)
fields.f3_s1 = ProtoField.string("proto.sf2-0", "Full name", nil, nil, nil, nil)
fields.f3_s2 = ProtoField.string("proto.sf2-1", "Nickname", nil, nil, nil, nil)
local f3_s3_valstr = { [0] = "away", [1] = "online", [2] = "busy", }
fields.f3_s3 = ProtoField.uint8("proto.sf2-2", "Status", base_DEC, f3_s3_valstr, nil, nil)
fields.payload = ProtoField.string("proto.payload", "Payload")

-- Dissector function --
function proto.dissector(buffer, pinfo, tree)

    local subtree = tree:add(proto, buffer())
    pinfo.cols.protocol = proto.name
    pinfo.cols.info = proto.description

    local packet = buffer():string()
    local def_fields = {true,true,true,}
    local def_subfields = { {}, {}, {true,true,true,}, }
    -- Delimiters --
    local field_delim = ":"
    local subfield_delim = "/"
    -- Field variables --
    local field = nil
    local field_delim_start = nil
    local field_delim_end = 1
    local field_start = 1
    local field_num = 1

    -- Search fields separated by the field delimiter --
    repeat
        if field_delim:len() > 0 then
            field_delim_start, field_delim_end = packet:find(field_delim, field_delim_end)
        end
        if field_delim_start ~= nil then
            field = packet:sub(field_start, field_delim_end)
            local subsubtree = nil
            if def_fields[field_num] then
                subsubtree = subtree:add(fields["f"..field_num],
                    buffer(field_start-1, field:len()))
            end
            -- Subfield variables --
            local subfield = nil
            local subfield_delim_start = nil
            local subfield_delim_end = nil
            local subfield_start = field_start
            local subfield_num = 1

            -- Search subfields separated by the subfield delimiter inside the field --
            repeat
                if not def_subfields[field_num] then break end
                if subfield_delim:len() > 0 then
                    subfield_delim_start, subfield_delim_end =
                        packet:find(subfield_delim, subfield_start, field_delim_start-1)
                end
                if subfield_delim_start ~= nil and subfield_delim_end < field_delim_start then
                    subfield = packet:sub(subfield_start, subfield_delim_start-1)
                    if def_subfields[field_num][subfield_num] then
                        subsubtree:add(fields["f"..field_num.."_s"..subfield_num],
                            buffer(subfield_start-1, subfield:len()))
                    end
                    subfield_num = subfield_num + 1
                    subfield_start = subfield_delim_end + 1
                end
            repeat
                if not def_subfields[field_num] then break end
                if subfield_delim:len() > 0 then
                    subfield_delim_start, subfield_delim_end =
                        packet:find(subfield_delim, subfield_start, field_delim_start-1)
                end
                if subfield_delim_start ~= nil and subfield_delim_end < field_delim_start then
                    subfield = packet:sub(subfield_start, subfield_delim_start-1)
                    if def_subfields[field_num][subfield_num] then
                        subsubtree:add(fields["f"..field_num.."_s"..subfield_num],
                            buffer(subfield_start-1, subfield:len()))
                    end
                    subfield_num = subfield_num + 1
                    subfield_start = subfield_delim_end + 1
                end
            end
        end
    end

```

```

        elseif subfield_num > 1 and def_subfields[field_num][subfield_num] then
            subfield = packet:sub(subfield_start, field_delim_start-1)
            subsubtree:add(fields["f"..field_num.."s"..subfield_num],
                buffer(subfield_start-1, subfield:len()))
        end
    until subfield_delim_start == nil or subfield_delim_end >= field_delim_start

    field_num = field_num + 1
    field_start = field_delim_end + 1
    field_delim_end = field_delim_end + 1
end
until field_delim_start == nil

-- Handle the payload --
local payload_delim = "\r\n"
local payload_delim_start = nil
local payload_delim_end = nil
payload_delim_start, payload_delim_end = packet:find(payload_delim, 41)
local subdiss_id = buffer(0,2):uint()
local payload_start = 40
proto_dt:try(subdiss_id, buffer(payload_start, (payload_delim_end-1)-payload_start+1):tvb(),
    pinfo, tree)

end

-- Dissector registration --
local parent_dt = DissectorTable.get("tcp.port")
parent_dt:add(9876, proto)

```

## Appendix 3. Lua code of the Wireshark dissector for DEMO3 protocol

```

-- Dissector for DEMO3 --
local proto = Proto("DEMO3", "DEMO3 Subprotocol")

-- Protocol field definitions --
local fields = proto.fields
fields.f1 = ProtoField.string("proto.ch1", "Channel 1", nil, nil, nil, nil)
fields.f2 = ProtoField.string("proto.ch2", "Channel 2", nil, nil, nil, nil)
fields.f3 = ProtoField.string("proto.ch3", "Channel 3", nil, nil, nil, nil)
fields.f4 = ProtoField.string("proto.ch4", "Channel 4", nil, nil, nil, nil)
fields.payload = ProtoField.string("proto.payload", "Payload")

-- Dissector function --
function proto.dissector(buffer, pinfo, tree)

    local subtree = tree:add(proto, buffer())
    pinfo.cols.protocol = proto.name
    pinfo.cols.info = proto.description

    local packet = buffer():string()
    local def_fields = {true,true,true,true,}
    local def_subfields = { {}, {}, {}, {}, {} }
    -- Delimiters --
    local field_delim = ","
    local subfield_delim = ""
    -- Field variables --
    local field = nil
    local field_delim_start = nil
    local field_delim_end = 1
    local field_start = 1
    local field_num = 1

    -- Search fields separated by the field delimiter --
    repeat
        if field_delim:len() > 0 then
            field_delim_start, field_delim_end = packet:find(field_delim, field_delim_end)
        end
        if field_delim_start ~= nil then
            field = packet:sub(field_start, field_delim_end)
            local subsubtree = nil
            if def_fields[field_num] then
                subsubtree = subtree:add(fields["f"..field_num],
                    buffer(field_start-1, field:len()))
            end
            -- Subfield variables --
            local subfield = nil
            local subfield_delim_start = nil
            local subfield_delim_end = nil
            local subfield_start = field_start
            local subfield_num = 1

            -- Search subfields separated by the subfield delimiter inside the field --
            repeat
                if not def_fields[field_num] then break end
                if subfield_delim:len() > 0 then
                    subfield_delim_start, subfield_delim_end =
                        packet:find(subfield_delim, subfield_start, field_delim_start-1)
                end
                if subfield_delim_start ~= nil and subfield_delim_end < field_delim_start then
                    subfield = packet:sub(subfield_start, subfield_delim_start-1)
                    if def_subfields[field_num][subfield_num] then
                        subsubtree:add(fields["f"..field_num.."s"..subfield_num],
                            buffer(subfield_start-1, subfield:len()))
                    end
                    subfield_num = subfield_num + 1
                    subfield_start = subfield_delim_end + 1
                elseif subfield_num > 1 and def_subfields[field_num][subfield_num] then
                    subfield = packet:sub(subfield_start, field_delim_start-1)
                    subsubtree:add(fields["f"..field_num.."s"..subfield_num],
                        buffer(subfield_start-1, subfield:len()))
                end
            repeat
                subfield_start = subfield_delim_end + 1
            until subfield_delim_end == nil
        end
        field_start = field_delim_end + 1
        field_num = field_num + 1
    until field_delim_end == nil
end

```

```

        end
        until subfield_delim_start == nil or subfield_delim_end >= field_delim_start

            field_num = field_num + 1
            field_start = field_delim_end + 1
            field_delim_end = field_delim_end + 1
        end
        until field_delim_start == nil

            -- Handle the payload --
            local header_delim = "\r\n"
            local header_delim_start = nil
            local header_delim_end = nil
            header_delim_start, header_delim_end = packet:find(header_delim)
            subtree:add(fields.payload, buffer(header_delim_end))

        end

        -- Dissector registration --
        local parent_dt = DissectorTable.get("demo2.port")
        parent_dt:add(6679, proto)

```