A Tool to Automate Generation of Wireshark Dissectors for a Proprietary Communication Protocol

Eric Golden and John W. Coffey
Department of Computer Science
The University of West Florida
Pensacola, FL. 32514

ABSTRACT

Wireshark is an open source packet analyzer that can be used to analyze and troubleshoot a variety of types of network data. Wireshark dissectors are used to break down packets into their constituent components, a critical part of the process of analysis. Developing a Wireshark dissector for a non-standard communication protocol can be a difficult and time-consuming task. Manually specifying Wireshark dissectors is an error-prone process that should be avoided. When the communication protocol in question is relatively simple, an automatically generated dissector might accomplish the main task of decoding network traffic in Wireshark, without the need for difficult manual implementation. This paper describes the development and use of an automatic Wireshark dissector generator.

Keywords: packet analyzer, Wireshark, communication protocol, dissectors

1. INTRODUCTION

Companies that develop or utilize network communication technologies often have a critical need to be able to examine and analyze network traffic. Whether looking for malicious content, implementing new protocols, debugging new or existing technologies, or improving efficiencies, network analysis tools are critically important. A variety of tools are available for the analysis of packets sent with standard network protocols, but the issue becomes more complicated when companies have a need to develop their own proprietary protocols.

Wireshark is an open source packet analyzer that can be used to analyze and troubleshoot a variety of network data including Ethernet, IEEE 802.11, PPP/HDLC, ATM, Bluetooth, USB, Token Ring, Frame Relay, FDDI, and others [1]. Wireshark runs on a variety of platforms and can perform a variety of useful functions for the analysis of packets being analyzed. For instance, it can be used to analyze packets transmitted in any of several hundred protocols, it has a variety of filters that can be applied, it can act upon compressed files, it supports several decryption schemes, and it has a variety of options for output format.

Wireshark relies on the specification of capture types and protocol-specific dissectors to be able to analyze network traffic. Specification of a dissector is complex and time-consuming, and it requires substantial technical capability. The goal of the current work is to develop a standard XML specification or vocabulary for a dissector and to create a proof-of-concept software program to parse the XML file and produce a Wireshark dissector. The Python programming language was chosen for the implementation of the dissector generator.

The rest of this article contains a description of related literature, a detailed description of the problem to be addressed, design and implementation issues that arose, and results that have been achieved. The paper closes with a discussion of further enhancements to the dissector.

2. RELATED LITERATURE

Literature is available describing various uses of Wireshark on standard communication protocols. For instance, Walberg [3] describes a use of Wireshark to analyze VoIP problems. The work he describes illustrates potential complexities in the problem since VoIP has separate protocols for signaling (connection establishment) and voice. He also describes additional difficulties due to the real-time nature of voice traffic. The various protocols require different dissectors for the constituent parts. Trapp's article [4] describes how Wireshark can be used to monitor network traffic on a smartphone, a non-trivial task since monitoring phone message traffic is generally tightly controlled. Trapp's article goes on to describe using Wireshark's capabilities to decode SSL certificates.
Wireshark is one of several network analysis tools with varying capabilities and degree of generality, and varying cost of use. The following discussion distills information from a summary that appears in Viswanathan et al [5], splunk [6] provides a range of capabilities for the analysis of any type of ascii-based data. Event Correlator [7] is used for the real-time matching of events based upon rules in a configuration file. Bro [8] is used for high-speed, real-time monitoring of packets. Wireshark is useful for the analysis of these standards-based, packet-based network data, but the need in the current problem is for the ability to analyze custom protocols.

One tool has been identified that enables the user to generate arbitrary communication protocols: Wireshark Generic Dissector [9]. This tool uses a distinctly C-like definition of a message structure to generate a Wireshark dissector. The main disadvantage to this type of specification is that the user must learn the special syntax of its input language, which is at the same level of abstraction as an equivalent structure in C, providing relatively little benefit over directly implementing a dissector at the code level. The goal of the current work is to create a system with the generality of the Wireshark Generic Dissector, but greater ease of use.

3. WIRESHARK PROCESSING

The process of dissecting a packet starts with the Frame dissector which processes the entire packet to remove its data [2]. The frame dissector passes the data on to the lowest-level data dissector. As an example, if the packet is Ethernet, the Ethernet dissector extracts the Ethernet header. The remaining data is then passed on to the next lowest-level dissector in the network communication protocol stack. The action of each dissector is to extract, decode, and display the data that is processed at the current level in the stack.

While Wireshark is loading packets from a file, each packet is dissected. Wireshark initially tries to detect the packet type. After detecting a known type, it extracts as much information from the packet as possible. Wireshark enables the user to select a specific packet for analysis. After the user has selected a specific packet from the list of available ones, the current packet is dissected again. Wireshark extracts all the information that it can from the current packet. The information is displayed in a different region of the interface, so multiple dissectors are needed for a single packet.

4. THE CURRENT PROJECT

This paper reports on a project to develop an open-source application written in Python which reads in an XML specification for a network communication protocol and generates C source code files that comprise a directly compilable Wireshark dissector plugin. Such a tool might make producing basic Wireshark dissectors for proprietary or experimental communication protocols almost as simple as defining the protocol in the first place. Several aspects of the problem exist including the creation of:

- an XML vocabulary defining the specification language
- the dissector generator
- several example specifications for different protocols for test purposes

Since this work is leading to a deployable product for a company, a user's manual was also developed.

The Problem

A significant portion of time on many projects developed by communications companies is spent developing and maintaining Wireshark dissectors for various proprietary communication formats over various media. The ability to develop custom dissectors becomes essential to the development and maintenance of such company's products, and hence, a strategic capability.

The goal of automating the Wireshark dissector generation capability is to reduce man-hours spent and complexity involved in developing and maintaining Wireshark dissectors, making their generation possible even for non-developer stakeholders in projects with proprietary communication protocols. The company for which this capability was developed provided input on the development process and sample protocols with which to test the application (although it is likely that many of these will not be released with the application).

The Implementation

The goal of this work was to implement an extensible Python-based XML-to-Anything generator. The program, named Transmute, is a Python3.x application that accepts an XML specification of a network communication protocol, and outputs source code for consumption by other tools. Currently, only Wireshark is supported, but other applications can be included by adding new plugins.
The base product was created over the course of five months by a single developer, and consists of approximately 2 KLOC, of which about 700 belong to the Wireshark-specific portion of the application. The relatively short development period culminated in a product that succeeded in its core purpose, as described above. The development was greatly accelerated by leveraging the Python programming language.

Using Python as the implementation language provided a number of benefits over other considered languages such as C++ or Java. Foremost, Python provides a platform-independent execution context with a rich feature set that can be relied upon anywhere the application could be expected to run. For instance, the XML parser used by the application is included with the standard distribution of Python, whereas a comparable parser for C++ would require including non-standard libraries, which may not be available for some platforms. Next, since almost all of the input and output data are expected to be string data, a scripting language seemed most appropriate, and Python provides a robust set of string operations. Finally, since Python is part of the build process for Wireshark, it can be assumed with reasonable certainty that Python is present in the development environment of anyone who might need to create a Wireshark dissector.

The benefit of an XML specification in this case is that it is instantly human-readable, and can be provided in almost the same manner as the high-level description document that defines the protocol. This high-level representation minimizes the risk of errors in translating the specification, and likely reduces the amount of time that would be required to find errors in such a transcription.

The development process required ongoing contact with the company for which the capability was being developed to ensure that the requirements were adequately captured, and to get feedback about the application’s specification and behavior. The software was developed in increments that were tested against real protocols, and the analyses that had been derived from use of the protocols.

This development effort followed an iterative software engineering pattern. The tool was tested on both Windows- and Ubuntu Linux-based operating systems. The initial effort involved developing an XML schema to use as the core of the protocol definition and decoding process, followed by the decoding portion of the Python application. The application was structured into modules with distinct responsibilities. The application is controlled via command line arguments and run without user interaction.

5. RESULTS

The XML Vocabulary

A standardized Wireshark namespace does not exist for XML. Consequently, one was created that contains the Wireshark-specific portions of the interface. This vocabulary includes provisions to expose dissector tables, to register protocols, dissector tree entries and groupings, etc. It is anticipated that future versions of the application will include additional namespaces that define specific extensions for other generators (e.g. C structures, C++ classes, Python classes, etc). This strategy ensures that plugins need not rely on co-opting tags intended for behaviors specific to another output in their extension to the application.

The protocol specification was largely informed by the way in which Wireshark presents a detailed packet dissection: a frame is divided into hierarchically layered protocols, each divided into a header and a trailer and different messages where appropriate; each of these is then further divided into specific fields. These tags were left deliberately general, so as to form a common vocabulary for describing a protocol in detail, regardless of its ultimate use. This left the vocabulary without any Wireshark-specific terms in the base set, so that behavior could be captured entirely within a dedicated XML namespace.

Degree of Automation Achieved

The initial version of the application provided a reasonable degree of automation for most target protocols. For very simple protocols, Transmute provides a useful, if rudimentary, Wireshark dissector that easily satisfies almost every use-case for only a small effort. For some protocols, Transmute will produce a dissector that requires some hand-editing by a competent developer, but this is usually limited to the suppression of an extraneous variable. Most uses of the application are likely to fall into one of these two categories. However, for even moderately complex protocols, the process of describing the protocol with the XML vocabulary can quickly surpass the effort required to implement a dissector directly, especially since some hand-editing of the C output is often necessary to capture protocol details that are not currently describable using the existing XML vocabulary.
Impact on Company Operations

After initial delivery, it was determined the product produced provided only part of the solution the company needed, although the other requirements were out of scope for the application and could be better solved by separate systems. Furthermore, while only software engineering staff make use of the product currently, future enhancements should make it accessible to other technical staff to use it as well. The role of Transmute in reducing the time required to produce Wireshark dissectors should improve as more features are added to the product. These features will eventually include the aforementioned C structure and C++ class generation, but also simple application generators to both originate and monitor packets in real time. These extensions could save tens to hundreds of man-hours per project, resulting in significant cost reduction and increased capability in the long term.

6. CONCLUSIONS

This work demonstrated the feasibility of creating a generic Wireshark dissector generator and identified some of the attendant difficulties. The creation of an XML vocabulary to describe the dissector makes the tool general and extensible, and was a significant focus of the work. The automation features of Python made its choice as the programming language a good one. The user base for the software is technical personnel who would be expected to know parameter values required.

The initial version of the software improved the efficiency of generating dissectors and was deemed a success. It is anticipated that enhancements to the initial version will further improve its utility. Possible enhancements or additional features include generating C structures and/or C++ classes from the XML specification. Such enhancements will make this tool useful in the automatic generation of other tools. Complex features, such as packet reassembly, Wireshark conversations, or other advanced features of Wireshark were beyond the scope of the initial effort. However, they are feasible and they would be highly desirable features in subsequent releases.

7. REFERENCES


