Using Attack Injection on Closed Protocols

João Antunes Nuno Neves Paulo Verissimo
LASIGE, Departamento de Informática,
Faculdade de Ciências da Universidade de Lisboa, Portugal
{jantunes,nuno,pjv}@di.fc.ul.pt

***Abstract*—Many network servers rely on the correctness and security of closed protocols. However, the unavailability of the protocol specification hinders any attempt to adequately test the implementations of that protocol. The paper addresses this problem by complementing an attack injection methodology with a protocol reverse engineering component. We introduce a new approach to automatically infer the message formats and the protocol state machine based only on network traces, without requiring access to the source code or binaries.**

***Keywords*-protocol reverse engineering; attack injection;**

I. Introduction

Commercial off-the-shelf network components frequently rely on closed protocols for their execution, preventing them from being adequately analyzed by third parties. Even in applications based on standard protocols, sometimes the developers add new features to differentiate their products from competing solutions, which cause reasonable changes to the underlying message formats and interactions. These extensions are often undocumented or poorly described, creating sizable parts of the functionality (and therefore, of the code) that is hard to assess from a security standpoint. Without the availability of the protocol specification, the correctness of any component implementing the protocol, either directly or by calling other modules, becomes difficult to scrutinize and depends on the testing activities carried out by the original programmers.

Attack injection is a methodology and tool for discovering security vulnerabilities, which generates and injects a large number of attacks against the network components being analyzed [1], [2]. This tool requires a specification of the protocol to be manually introduced to derive meaningful test cases, which have been shown to be effective at finding previously unknown flaws. Nonetheless, this approach has a few limitations: it can only be applied to open protocols (e.g., presented in a RFC) and standard features; and, it is time consuming and error prone to incorrect interpretations of the documentation. In this paper, we introduce a new approach to automatically infer the protocol specification, which is based on the analysis of collected network trace data, and that can be used to extend attack injection to support closed protocols.

Recently, a few researchers started to investigate protocol reverse engineering following two main approaches. In the first one, the tools closely monitor the program's execution while processing a single protocol message. An execution trace is then used to identify and to correlate the relevant sections of the code responsible for processing and parsing specific parts of the message (e.g., the message fields) [3], [4]. These solutions typically use a single protocol interaction to infer the format of the respective message type, but there have been extensions to improve the accuracy of message format inference [5] and to derive a protocol state machine [6]. These solutions are restricted by the limitations of the dynamic tainting tool, such as being operating system dependent and requiring access to an implementation of the protocol (besides the network traces).

The other approach assumes no access to the source code or binary, relying solely on a substantial sample of the protocol interaction. Protocol informatics applied concepts from bioinformatics to construct a rough description of message formats from the network traces [7]. The tool employs sequence alignment algorithms to reveal similarities and differences between messages. Discoverer performs clustering and type-based sequence alignment [8]. An initial clustering algorithm groups messages with similar sequences of text or binary tokens. Then, recursive clustering and sequence alignment techniques refine each cluster to produce more detailed message formats. However, neither of these solutions provides a complete protocol specification since they do not attempt to derive the protocol state machine.

II. Deriving a Protocol specification

At this stage, we are focusing on application protocols that are mainly used in network servers. We observed that several of these protocols are text-based (e.g., HTTP, Microsoft Messenger, SIP, IMAP, FTP), and therefore, we chose to target this type of protocols, thus taking advantage of how text fields are usually organized and delimited in a message. Our approach consists in two separate phases:

- A finite state machine (FSM) is iteratively constructed from the messages of the network traces. Each message is parsed and transformed into a sequence of symbols, and given to the automaton to process. New states and transitions are added in order accept the new sequence. Since this raw FSM only accepts the messages it processed, we next abstract and generalize the irrelevant parts of the automaton. By analyzing the frequency that each field appears in the traces, a generalization

algorithm is employed to merge transitions that are identified as individual instances of message parameters, therefore resulting in a FSM that is able to accept any instance of the messages present in the traces. Accordingly, each path in the resulting FSM corresponds to protocol message format.

- A state machine of the protocol is deduced from the network traces' sessions and from the inferred message formats. We start by dividing the network traces in individual client server protocol sessions. Next, a state machine is built from the sequence of message types in each session. Then, we identify similar states as states in automaton that are reached under similar conditions, and merge them. By producing a FSM that captures the potential causal relationships between the different types of messages, we infer the protocol state machine.

The obtained protocol specification can be used to generate messages that are accepted by components implementing the protocol, either the client or the server, depending on which direction of the message flow that was used. Moreover, other applications using the same protocol will also accept messages generated from the inferred specification.

However, since our approach relies on a sample of the protocol interaction to generalize the message formats and to derive the protocol state machine, it may result in an incomplete protocol specification or in an over-generalization (e.g., a biased field history may incorrectly induce a generalization on that field). Naturally, these problems are inherent to the quality of the network traces, and they might impact on the quality of the generated attacks.

Figure 1 depicts the architecture of an attack injection tool that uses our approach to derive the protocol specification. The tool can passively listen to the network server's incoming traffic to obtain the traces and to calculate the protocol state machine. Then, its attack generation algorithms can use that specification to create test cases with (malicious) protocol messages. The attacks can also be injected in any state of the protocol (correct or even unexpected), since the attack injector can manipulate the protocol connection to take it to the desired state. The monitor component of the attack injector is able record and collect an execution trace of the server for each attack injection, that will later be analyzed. Suspicious behavior on the execution trace, such as an unusual set of system calls, a large resource usage, or a bad return error code, usually indicate that the server has experienced some fault while processing that particular attack. The offending attacks can then be given to the developers to reproduce the anomalies and to confirm the existence of vulnerabilities.

III. Conclusions

This paper addresses the problem of testing applications that rely on closed protocols. Attack injection has been used to discover vulnerabilities on network servers by

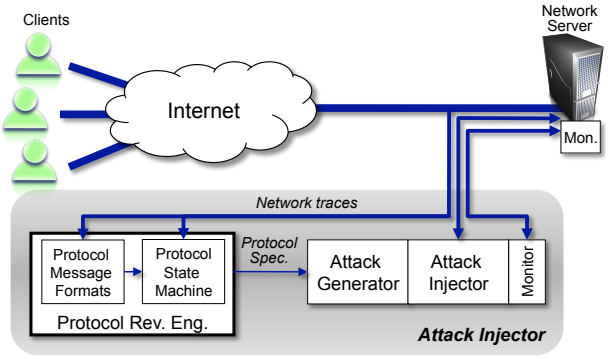

Figure 1. Attack injector with the protocol reverse engineering component.

generating test cases from the protocol specification. Our protocol reverse engineering component can automatically infer the protocol message formats and the state machine of the protocol by passively analyzing the incoming network traffic. The attack injector can then use this information to construct (malicious) protocol messages by its attack generation algorithms.

Acknowledgements: This work was partially supported by the FCT through the Multi-annual and the CMU-Portugal Programmes, and the project PTDC/EIAEIA/100894/2008 (DIVERSE).

References

[1] N. Neves, J. Antunes, M. Correia, P. Verissimo, and R. Neves, "Using attack injection to discover new vulnerabilities," in *Proceedings of the International Conference on Dependable Systems and Networks*, Jun. 2006.

[2] J. Antunes, N. Neves, M. Correia, P. Verissimo, and R. Neves, "Vulnerability removal with attack injection," IEEE Transactions on Software Engineering, accepted for publication.

[3] J. Caballero, H. Yin, Z. Liang, and D. Song, "Polyglot: Automatic extraction of protocol message format using dynamic binary analysis," in *Proceedings of the ACM Conference on Computer and Communications Security*, 2007.

[4] Z. Lin, X. Jiang, D. Xu, and X. Zhang, "Automatic protocol format reverse engineering through context-aware monitored execution," in *Proceedings of the Network and Distributed System Security Symposium*, 2008.

[5] G. Wondracek, P. Comparetti, C. Kruegel, E. Kirda, and S. Anna, "Automatic network protocol analysis," in *Proceedings of the Annual Network and Distributed System Security Symposium*, 2008.

[6] P. M. Comparetti, G. Wondracek, C. Kruegel, and E. Kirda, "Prospex: Protocol specification extraction," in *IEEE Security and Privacy*, 2009.

[7] M. A. Beddoe, "Network protocol analysis using bioinformatics algorithms," 2005, http://www.4tphi.net/∼awalters/PI/PI.html.

[8] W. Cui, J. Kannan, and H. Wang, "Discoverer: automatic protocol reverse engineering from network traces," in *Proceedings of the USENIX Security Symposium*, 2007.