Protocol State Fuzzing of TLS

Ben Broadhead, Isaac Freeland and Charlie Shoup

***Abstract*—This paper talks about the use of protocol state fuzzing on OpenSSL implementations of TLS. Protocol state fuzzing is a semi-automated version of random testing. This is different from traditional fuzzing where random inputs are sent to a program. Protocol state fuzzing sends correctly formated packets in random orders at a protocol in order to move through the protocols states. TLS is the foundation of secure network communication. Most devices connected to networks require secure and encrypted communication. Ensuring that TLS operates in a secure manner is of utmost importance. We use state machine learning to test different versions of OpenSSL's implementation of TLS.**

***Index Terms*—Fuzzing, TLS, OpenSSL**

I. Introduction

Computer networks are a fundamental part of modern society. Almost every person in the developed world is connected together via the Internet. As the Internet has become more integrated into everyday life, the need for secure communication has become a necessity. This integration comes with an every increasing complexity that must be understood and debugged by software engineers.

In order to test increasingly large and complex programs, a semi-automated black box testing approach known as protocol state fuzzing can be used. Fuzz testing can generate large amounts of inputs, is mostly automated, and can develop unique series of inputs that a software tester might not think of while manually writing a test suite. The point of protocol state fuzzing is to send correctly formated messages in all possible orders. This means that there will be at least one path taken through the protocol that ends in an accepted state (the "correct path"). Once again, this testing method is black box and therefore can be tested on a verity of implementations of the same protocol. After the fuzz tester provides each input to a protocol the results are recorded. These results are not necessarily bugs, but can be crashes, reject or error messages, or accept messages. Using a state learning algorithm named L-star (L*) [1] the protocol fuzz tester generates a state graph by keeping track of what messages get returned after what messages have been sent. Different implementations and different versions of a protocol can produce very different state graphs. However, the "correct path" through each version and implementation should remain the same.

Transport Layer Security (TLS) finds widespread use in thousands of different applications, most notably in HTTPS. Having a secure, bug-free connection between a web browser and a server is essential to protecting the integrity and privacy of users. Upper layer applications such as websites rely on lower level network layers for integrity protection and security from adversaries. There are many different implementations of TLS available though a variety of software libraries. We chose to test OpenSSL as it finds widespread use on many servers and is open-source. We make use of protocol state fuzzing to check for unexpected jumps between states in OpenSSL. [2]

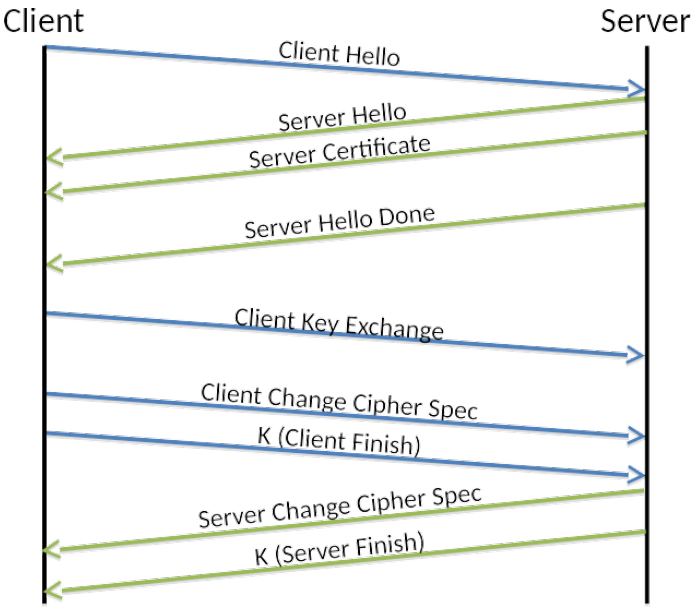

Figure 1. Client server message exchange using TLS. TLS is implemented in a variety of ways including with OpenSSL. This project only tests the server side of OpenSSL.

A. Protocol State Fuzzing Vs. Traditional Fuzzing

To understand this project, it is important to know about different types of fuzzing. For this project there are two important fuzzing methods to be aware of: Traditional fuzzing and protocol state fuzzing.

Traditional fuzzing is the process of generating randomized inputs to find bugs in programs. By creating a large number of malformed messages, it is possible to see if a program will return unexpected behavior [3]. Traditional fuzzing allows a user to find memory leaks and crash states within a program. Fuzzing is an old method of testing programs, dating all the way back to the 1950s when computer still used punch cards [4].

Protocol state fuzzing takes known protocol messages and changes the order in which these messages are sent. This method is useful for finding the states of a protocol and how each state relates to one another. State diagrams can be produced giving a user visual perception of the protocol. Protocol state fuzzing can detect if states are being skipped or accessed in undesired ways. Protocol state fuzzing was

first introduced in 1988 by Barton Miller at the University of Wisconsin - Madison [5].

Both of these fuzzing methods are useful for different reasons. Protocol state fuzzing builds on top of traditional fuzzing. Adding an element of traditional fuzzing to the state learning tools would further increase the reliability of our results. The addition of traditional fuzzing would be the next thing to add if one desired to continue this research. Due to limited time constraints, only protocol state fuzzing used.

II. Related Work

Protocol state fuzzing is not a unknown approach to finding vulnerabilities in encryption and authentication protocols. In the case of TLS, dutch security researcher Joeri de Ruiter has done a large portion of the existing work. De Ruiter's approach involved using Java code on top of the LearnLib [6] framework to create state machines which modeled the behavior of common TLS implementations. Once a state machine is initialized, is can be tested with an 'alphabet' containing 'letters' representing various transmitted messages. LearnLib uses the given alphabet to test the state machine model until a specified depth has been reached. While the test is being performed, the Java container logs all of the reached states as well as the edges (letters from the given alphabet) which are found to connect them.

De Ruiter's work is not current. Despite his creation of a solid framework from which to perform protocol fuzzing, the project has not been known to have disclosed any vulnerabilities within the last year. In this time, the TLS model underpinning OpenSSL has been updated. Due to the nature of implementation vulnerabilities, one series of bug fixes may not stymie attackers for long and may in fact create more paths to spurious states than they fix. Unluckily, there appears to be little else published in the interim leaving a void of public knowledge regarding state errors in newer TLS implementations.

III. Adversary model

Anyone with a computer and an ability to view and inject packets into a network can be considered an adversary. Protocol state fuzzing can reveal a wide verity of bugs in a protocol. In this case of TLS this can mean that an adversary could potentially perform attacks like session hijacking all the way to improper authentication or data transfer without authentication.

IV. Methodology

In order to test current OpenSSL implementations, it was necessary to first verify that prior research could be replicated. If prior research was shown to be repeatable, the next step would be to apply the learning algorithm to newer versions of OpenSSL. If spurious states (possible vulnerabilities) were found, the alphabet would be modified with custom letters which could lead the found state to exhibit unfavorable behavior (edges leading to otherwise unaccessible states).

V. Implementation

The first step in being able to test OpenSSL's implementation of TLS was making sure we understood how to use the fuzzing tools. This required reimplementing Joeri de Ruiter's state graphs for old versions of OpenSSL [2]. To do this we tested the state learning tools against OpenSSL 1.0.1j. In De Ruiter's paper, a state graph for this version of OpenSSL is provided, show in figure 2. Based on runs of the protocol fuzzer on multiple operating systems running the same version of OpenSSL, it was possible to verify the De Ruiter analysis of 1.0.1j. A replicated version on OpenSSL 1.0.1j is seen figure 3. Various other runs were then conducted on OpenSSL variants with a standard alphabet, standard learning algorithm and varied depth settings.

In order to efficiently test protocols, a few concessions had to made. There was limited time for experimentation and setting the learner to produce a graph with a depth greater than 7 was computationally inefficient on laptop computers. Throughout all of the tests, the learning algorithm from the De Ruiter research was left unchanged however changes were made to the Java container in order to improve the logging capability. The largest graph generated had a depth of 7 which took 12 hours to generate. Tests were performed in parallel on multiple laptops and results were studied visually for anomalous behavior. Finally, since replicating the results from version 1.0.1.j confirmed the test method to be valid, OpenSSL 1.1.0g was tested for vulnerabilities.

OpenSSL 1.1.0g, the most current OpenSSL version as of December 2017, was tested for vulnerabilities. A state graph of the version is show in figure 4. This graph was generated using the same settings as the 1.0.1j version, but yielded less states. This version showed that less spurious states exist in newer version of OpenSSL, suggesting that it might be more secure. Version 1.0.1j had four spurious states, while 1.1.0g had only one state.

In the various figures, green lines show the correct path to a valid exchange with an OpenSSL server. In the event of a invalid state, the connection should close. States (numbered bubbles) not linked by green lines in any form are considered to be spurious.

It is important to note that the implementation described here is far from perfect. It is unlikely that much improvement could have been made to the L* learning algorithm chosen however time or more computational ability would allow graphs of greater depth to be efficiently generated. As well, it would be helpful to display the generated graph in real time so that bugs could be logged as a run progressed, rather than solely upon completion.

VI. Conclusion

Protocol state fuzzing is a useful tool for finding security flaws or unknown states in various protocols. Our worked expanded on previous fuzzing of older versions of OpenSSL. Testing against old versions of OpenSSL was necessary to make sure our method of testing was valid. After replicating the results of previously tested versions of OpenSSL, it was

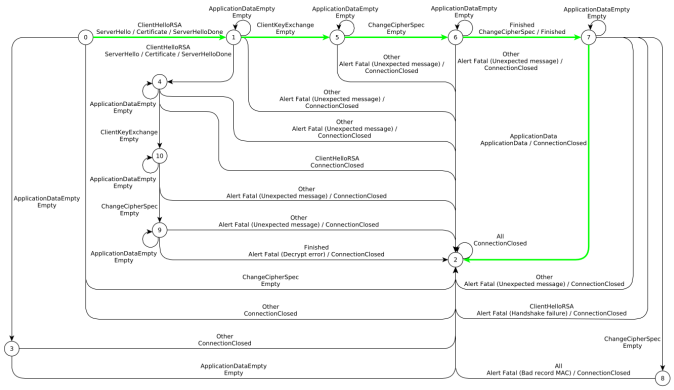

Figure 2. Known state graph of OpenSSL 1.0.1j [2]

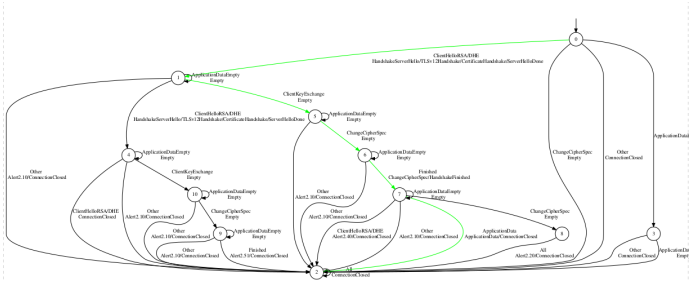

Figure 3. Replicated state graph of OpenSSL 1.0.1j. This state graph shows the same number of states and transitions as that done in Joeri de Ruiter's tests. Replicating known state graphs was essential in understanding how to operate the state learning tools.

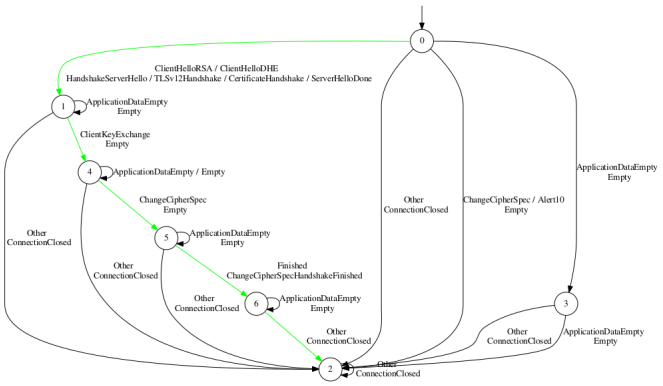

Figure 4. State graph of OpenSSL 1.1.0g. This graph shows less spurious states that that of OpenSSL 1.0.1j. This version only contained one spurious state, compared to four spurious states in version 1.0.1j

possible to check current versions for potential vulnerabilities. Our work cannot prove with certainty that OpenSSL is completely secure. Our work does support the notion that current versions of OpenSSL contain less spurious states than their predecessors, suggesting that they are more secure. The existence of a spurious state does not prove that a vulnerability exists. Extra inspection of these states is necessary to see if a threat does indeed exist. Due to limited time constraints, it is unknown if the spurious state of OpenSSL 1.1.0g is a cause for concern.

Throughout the research process, a few instances of anomalous behavior were observed. Although replication of prior results appeared to validate the methods used, running the StateLearner multiple times on the same hardware with the same hard-coded initial conditions could produce different graphs. It is likely that the reason for these different graphs has something to do with the L* algorithm backing the operations, however a conclusion was not reached. If the results from these runs can be trusted, than spurious states may be much more common than one might suspect. However, hardware specific initial conditions may play a greater role than previously expected.

A second instance of anomalous behavior which was observed is the number of states which are generated from runs with a depth greater than 7. For an undetermined reason, a very large number of states are generated, even though the runs do not have enough time to finish. There is no known function to merge states after a run, leading to some confusion as to where all of the states arise from. It is likely that this is a bug in Statelearner, as there is no reason for the number of spurious states to increase nearly exponentially when the depth is increased by a small factor. It is possible that these states represent waits, however only more runs with greater depth on previously studied implementations can say for sure.

Assuming that the spurious state in 1.1.0g is able to replicated, i.e. multiple runs on multiple machine can produce the same or similar graph with the same path. Further testing is necessary to confirm the vulnerability and it's implications. With more time, this approach to testing OpenSSL could be broadened. Traditional fuzzing could be implemented to test not only against know message formats, but also investigate the effects of carefully randomized data being sent from one of target states. In this case, it would likely be possible to discard the machine learning framework entirely and proceed using a packet crafting tool.

VII. Project Repository

All code and project files mentioned in this paper can be found at https://github.com/broadheadbenjamin/CS-6490

References

[1] D. Angluin, "Learning regular sets from queries and counterexamples," in *Information and Computation 75*, 1987, pp. 87–106.

[2] J. De Ruiter and E. Poll, "Protocol state fuzzing of tls implementations." in *USENIX Security Symposium*, 2015, pp. 193–206.

[3] P. Godefroid, M. Y. Levin, and D. Molnar, "Sage: whitebox fuzzing for security testing," *Queue*, vol. 10, no. 1, p. 20, 2012.

[4] "Fuzz testing and fuzz history," https://secretsofconsulting.blogspot.com/2017/02/fuzz-testing-and-fuzz-history.html, accessed: 2017-11-28.

[5] A. Takanen, "Fuzzing: the past, the present and the future," in *Actes du 7ème symposium sur la sécurité des technologies de l'information et des communications (SSTIC)*, 2009, pp. 202–212.

[6] "The learnlib project homepage," https://learnlib.de/, accessed: 2017-11-20.