Is Stateful Fuzzing Really Challenging?

Cristian Daniele
cristian.daniele@ru.nl
Radboud University
Nijmegen, Netherlands

ABSTRACT

Fuzzing has been proven extremely effective in finding vulnerabilities in software. When it comes to fuzz stateless systems, analysts have no doubts about the choice to make. In fact, among the plethora of stateless fuzzers devised in the last 20 years, AFL (with its descendants AFL++ and LibAFL) stood up for its effectiveness, speed and ability to find bugs. On the other hand, when dealing with stateful systems, it is not clear what is the best tool to use. In fact, the research community struggles to devise (and benchmark) effective and generic stateful fuzzers. In this short paper, we discuss the reasons that make stateful fuzzers difficult to devise and benchmark.

KEYWORDS

Stateless fuzzing, Stateful fuzzing, Software testing

1 INTRODUCTION

Fuzzing consists of sending many malformed (or slightly malformed) messages to a program in order to trigger crashes. Despite the approach being straightforward, clever engineering stunts (like code instrumentation) made the technique very effective and scalable. The pioneer of the approach is AFL [13] (and its successors AFL++ and the more recent LibAFL [6]), which, since 2013, has found plenty of bugs daily. The problem is that the above-mentioned tools are devised to tackle stateless systems (as explained in Section 2), i.e. systems that do not have any notion of sessions and do not need to implement any state model. Unfortunately, the vast majority of protocols — being stateful — need state models to work. The statefulness of these systems makes *all* the stateless fuzzers almost totally ineffective (as explained in Section 3). For this reason, the community developed fuzzers able to bypass or deal with the statefulness of the systems (as explained in Section 4).

2 STATELESS AND STATEFUL FUZZING

Fuzzers (both stateless and stateful) send millions of messages to the System Under Test (SUT) to trigger crashes or weird system behaviors [14]. To generate these messages cleverly, mutational-based fuzzers (often called smart evolutionary or grey-box), instrument the code to get feedback from the SUT about the quality of the messages sent. Basically, the fuzzer rewards messages that trigger new coverage in the SUT code.

The main difference between stateless and stateful fuzzers is that while the formers only need to mutate the single messages to (theoretically) get 100% coverage, the latter also need mutations over the order of the messages (often called trace) to (theoretically) get 100% coverage. For this reason, stateless fuzzers usually achieve poor results fuzzing stateful systems.

3 LIMITS OF STATELESS FUZZERS

Despite being possible to use stateless fuzzers over stateful systems, it is usually not a good choice. In fact, stateless fuzzers lack some abilities required to efficiently fuzz stateful systems. Namely, they cannot:

(1) send messages over the network. It is easy to implement, but requires modifing the fuzzer's code or using specific libraries [1];
(2) avoid the SUT from restarting after every message. AFL and all the AFL-based fuzzers restart the SUT after every message. This behaviour precludes the possibility of fuzzing deep states since, after the restart, the SUT starts again from the initial state;
(3) mutate the order of the messages sent to the SUT. Stateless fuzzers do not provide mutation functions on the trace level, making impossible mutations over the traces;
(4) focus on the most interesting state (or any state, altogether), as they do not have any notion of state models;
(5) observe the response of the server to steer the generation of the messages or infer the state model of the SUT.

The above-mentioned flaws highlight the need to devise fuzzers able to deal with stateful systems.

4 APPROACHES USED BY STATEFUL FUZZERS

Different approaches have been developed to fuzz stateful systems. While some of them try to bypass the stateful nature of the system; others try to deal with it.

4.1 Bypassing the statefulness of the systems

The research community came up with different approaches to using stateless fuzzers over stateful systems.

4.1.1 Bringing the SUT to a certain state. The simplest approach to bypass the statefulness of the systems is perhaps bringing the SUT to a certain state (by using a prefix) and then using a stateless fuzzer. For example, the *custom mutators* implemented by AFL++ easily allow prepending fixed prefixes to the messages to mutate.

In order to use a more systematic approach, it is possible to describe the prefixes through a grammar and use a grammar-based fuzzer. For a grammar-based fuzzer ([7], [5]) the statefulness of a system is not an issue as the grammar provided can describe both the structure of the messages (needed also to fuzz stateless systems) and the structure of the traces [4]. As already mentioned, the grammar of the state model is often *only* used to lead the SUT to a specific state. In other words, the fuzzers do not use the grammar for the mutations but to reach and tackle interesting[1] states. The reason behind this choice is straightforward: mutations over the

[1] In this scenario, the analyst decides whether a state is interesting or not.

content and the order of the messages will likely lead to chaotic trace generations.

4.1.2 Adding artificial loops. Another approach to bypass the statefulness of the systems consists of adding an artificial loop (in the SUT) that wraps the handling of the commands. This loop avoids the restarting of the SUT after every message and — very artificially — allows mutations over the traces. *AFL persistent mode* (originally devised exclusively stateless systems) allows such a behaviour. It allows annotating the SUT code with a *AFL_LOOP(n)* instruction to cycle a specific portion of code *n* times [2]. A drawback of the approach is the possibility of ending up in an unprofitable state (such as a state that merely handles errors). To avoid this problem, it is possible to include in the fuzzing dictionary (often called seed file) a message that brings back to the initial state.[2]

4.1.3 Using fuzzing targets. To bypass the statefulness of the systems, it is also possible to use *fuzzing targets*. Stateless fuzzers like LibFuzzer allow one to tackle portions of code by providing specific entry points to specific functions. Despite this approach theoretically working in fuzzing stateful systems, it can overlook all the bugs triggered by some sort of relation between the different messages in the same trace. For example, assume that the trace $< m_1, m_2, m_3 >$ triggers the bug. Thanks to the fuzzing targets, it is possible to fuzz specifically the piece of code parsing the message m_2. Unfortunately, this will overlook all the issues caused by weird relations between the different messages in the trace. Moreover, this approach makes it challenging to trigger the bug from the outside as the analyst has no clue about the structure of the whole trace. In fact, analysts would only know the single message that triggers a bug in that specific function.

4.1.4 Cram multiple messages into one. Another strategy to bypass the statefulness of the systems by using an AFL-based fuzzer is to cram several messages (separated by a delimiter) into one. The message would then be something like $M = m_1 \backslash n m_2 \backslash n m_3$, assuming the delimiter being the newline character ($\backslash n$). Doing this allows fuzzing deeper states but does not give any control over the mutations of the single messages. In fact, mutating the single messages would require ad hoc mutation functions able to chop the messages up. To the best of our knowledge, no fuzzer uses this approach.

It is worthwhile to note that all the above-mentioned approaches only solve the problem (2) described in Section 3 but still do not allow focusing on the more promising states or leveraging server responses.

4.2 Dealing with the statefulness of the systems

Other approaches ([11], [3], [9]), instead of trying to bypass the statefulness of the systems, try to deal with it by sending *entire traces* (and not single messages) to the SUT.

This allows the fuzzers to make mutations over single messages and also over their order. More advanced fuzzers also have modules to infer and take into account the state model of the SUT. Sending entire traces is slow. For this reason, some stateful fuzzers (like nyx-net [12]) introduced a technique called *snapshotting* that allows to *snapshot* the program in a certain state to be able to reach the same state more quickly in the future. Despite the technique aiming to make the fuzzing process faster, it often introduces such a huge overhead that makes re-sending the entire sequence of messages more convenient.

Unfortunately, the majority of these fuzzers are tailored to specific systems and require some tuning to work nicely on others. Typical modifications required are the writing of modules to parse protocols requests and responses [11], and SUT modifications to highlight state variables [3], getting rid of forks [8], sockets [1], and any cryptography whatsoever.

5 BENCHMARKING STATEFUL FUZZERS

Another issue when coming to stateful fuzzing regards their benchmarking. In fact, while for stateless fuzzers it is reasonable to benchmark them according to the code coverage they achieved, things get more complicated with the stateful ones. For stateful fuzzers, also the *state coverage* plays a crucial role since high coverage in the code does not imply high coverage in the state model. Unfortunately, monitoring the state coverage is challenging and requires the actual state model of the SUT to track which (and how many times) states have been fuzzed.

To the best of our knowledge, ProFuzzBench [10] is the only benchmark framework for stateful protocols. Despite being widely used to compare stateful fuzzers, it only compares fuzzers regarding *code coverage*. Useful extensions to the framework might be modules to monitor the state coverage and more fine-grained information about the number of messages sent and time in relation to the coverage. Monitoring the time needed to achieve a certain coverage only gives information about the speed of the fuzzer. On the other hand, coverage in relation to messages sent gives information about the ability of the fuzzer to craft interesting messages. Minimising the number of messages sent implies a deeper understanding of the structure of the SUT.

Another challenge when benchmarking stateful fuzzers regards the possible inconsistency of some stateful fuzzers. In fact, some fuzzers might be good at exploring the state model while others might excel in fuzzing single states, so the benchmarking can be highly biased by the nature of the system tested (i.e. SUT implementing simple or complex state models).

6 CONCLUSIONS

Despite the community being actively involved in devising stateful fuzzers, most of the approaches are not scalable but tailored to particular SUTs. The challenges presented in the paper show that *the* stateful fuzzer is still far away. The lack of mature and generic stateful fuzzers is partially justified by stateful fuzzing being a much younger field than stateless fuzzing.

Eventually, this paper sheds light on the challenges and techniques of stateful fuzzing, giving room for further research and future directions.

ACKNOWLEDGMENTS

This research is funded by NWO as part of the INTERSECT project (NWA.1160.18.301).

[2]The SUT needs to implement such a command; otherwise the analyst needs to implement it.

REFERENCES

[1] Seyed Behnam Andarzian, Cristian Daniele, and Erik Poll. 2023. Green-Fuzz: Efficient Fuzzing for Network Protocol Implementations. In *International Symposium on Foundations and Practice of Security*. Springer, 253–268.

[2] Anonymous. 2024. AFL*: A Simple Approach to Fuzzing Stateful Systems. (2024). anonymous preprint under review.

[3] Jinsheng Ba, Marcel Böhme, Zahra Mirzamomen, and Abhik Roychoudhury. 2022. Stateful greybox fuzzing. In *31st USENIX Security Symposium (USENIX Security 22)*. 3255–3272.

[4] Cristian Daniele, Seyed Behnam Andarzian, and Erik Poll. 2023. Fuzzers for Stateful Systems: Survey and Research Directions. *Comput. Surveys* (2023).

[5] Adam Doupé, Ludovico Cavedon, Christopher Kruegel, and Giovanni Vigna. 2012. Enemy of the State: A State-Aware Black-Box Web Vulnerability Scanner. In *21st USENIX Security Symposium (USENIX Security 12)*. 523–538.

[6] Andrea Fioraldi, Dominik Christian Maier, Dongjia Zhang, and Davide Balzarotti. 2022. LibAFL: A Framework to Build Modular and Reusable Fuzzers. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 1051–1065.

[7] Hugo Gascon, Christian Wressnegger, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. 2015. Pulsar: Stateful Black-Box Fuzzing of Proprietary Network Protocols. In *Security and Privacy in Communication Networks: 11th EAI International Conference, SecureComm 2015, Dallas, TX, USA, October 26-29, 2015, Proceedings*. Springer, 330–347.

[8] Marcello Maugeri, Cristian Daniele, Giampaolo Bella, and Erik Poll. 2023. Evaluating the Fork-Awareness of Coverage-Guided Fuzzers. *International Conference on Information Systems Security and Privacy - ICISSP* (2023).

[9] Roberto Natella. 2022. StateAFL: Greybox Fuzzing for Stateful Network Servers. *Empirical Software Engineering* 27, 7 (2022), 191.

[10] Roberto Natella and Van-Thuan Pham. 2021. Profuzzbench: A Benchmark for Stateful Protocol Fuzzing. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 662–665.

[11] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. 2020. AFLNet: A Greybox Fuzzer for Network Protocols. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 460–465.

[12] Sergej Schumilo, Cornelius Aschermann, Andrea Jemmett, Ali Abbasi, and Thorsten Holz. 2022. Nyx-Net: Network Fuzzing with Incremental Snapshots. In *Proceedings of the Seventeenth European Conference on Computer Systems*. 166–180.

[13] Michal Zalewski. 2014. American Fuzzy Lop (AFL). https://lcamtuf.coredump.cx/afl

[14] Xiaogang Zhu, Sheng Wen, Seyit Camtepe, and Yang Xiang. 2022. Fuzzing: A Survey for Roadmap. *ACM Computing Surveys (CSUR)* 54, 11s (2022), 1–36.