Identifying Vulnerabilities in SCADA Systems via Fuzz-Testing

Rebecca Shapiro, Sergey Bratus, Edmond Rogers, Sean Smith

Chapter 5

IDENTIFYING VULNERABILITIES IN SCADA SYSTEMS VIA FUZZ-TESTING

Rebecca Shapiro, Sergey Bratus, Edmond Rogers and Sean Smith

Abstract Security vulnerabilities typically arise from bugs in input validation and in the application logic. Fuzz-testing is a popular security evaluation technique in which hostile inputs are crafted and passed to the target software in order to reveal bugs. However, in the case of SCADA systems, the use of proprietary protocols makes it difficult to apply existing fuzz-testing techniques as they work best when the protocol semantics are known, targets can be instrumented and large network traces are available. This paper describes a fuzz-testing solution involving LZFuzz, an inline tool that provides a domain expert with the ability to effectively fuzz SCADA devices.

Keywords: Vulnerability assessment, SCADA systems, fuzz-testing

1. Introduction

Critical infrastructure assets such as the power grid are monitored and controlled by supervisory control and data acquisition (SCADA) systems. The proper functioning of these systems is necessary to ensure the safe and reliable operation of the critical infrastructure – something as simple as an input validation bug in SCADA software can leave an infrastructure asset vulnerable to attack. While large software development companies may have the resources to thoroughly test their software, our experience has shown that the same cannot be said for SCADA equipment manufacturers. Proell from Siemens [19] notes that random streams of bytes are often enough to crash SCADA devices.

Securing SCADA devices requires extensive testing for vulnerabilities. However, software vulnerabilities are often not well understood by SCADA developers and infrastructure experts, who may themselves not have the complete protocol documentation. Meanwhile, external security experts lack the SCADA knowledge, resources and access to run thorough tests. This is a *Catch-22* situation.

Fuzz-testing is a form of security testing in which bad inputs are chosen in attempt to crash the software. As such, it is widely used to test for security bugs in input validation as well as in application logic. However, applying fuzz-testing methodologies to secure SCADA devices is difficult. SCADA systems often rely on poorly understood proprietary protocols, which complicates test development. The time-sensitive, session-oriented nature of many SCADA environments makes it impossible to prime a fuzzer with a large capture. (Session data is only valid for a short time and is often rejected out of hand by the target thereafter.) Furthermore, many modern fuzzers require users to attach a debugger to the target, which is not always possible in a SCADA environment. What is needed is a fuzzer than works inline.

This paper describes LZFuzz, an inline fuzzing tool that enables infrastructure asset owners and operators to effectively fuzz their own equipment without needing to modify the target system being tested, and without having to expose their assets or pass proprietary information to external security evaluators.

2. Fuzzing Overview

Barton Miller, the father of fuzz-testing, observed during a thunderstorm that the lightning-induced noise on his network connection caused programs to crash [15]. The addition of randomness to inputs triggered bugs that were not identified during software testing. Upon further investigation, Miller discovered that the types of bugs triggered by fuzzing included race conditions, buffer overflows, failures to check return code and `printf`/format string problems. These bugs are often sources of software security vulnerabilities [14]. Most modern software undergoes aggressive input checking and should handle random streams of bytes without crashing. Consequently, modern fuzz-testing tools have become more selective in how they fuzz inputs.

Whether or not data has been fuzzed, there usually are multiple layers of processing that the data has to undergo before it reaches the target software's application logic. Application logic is the soft underbelly of software – penetrating it greatly increases the likelihood of compromising the software. Fuzzed inputs trigger bugs only if they are not rejected by one of the processing layers before they get to the application logic. Therefore, a fuzzer must generate inputs that are clean enough to pass all the processing layer checks, but that are sufficiently malformed to trigger bugs in the application logic.

The most successful fuzzers create fuzzed inputs based on complete knowledge of the layout and contents of the inputs. If a fuzzer is given information on how a specific byte will be interpreted, it can manipulate the byte in ways that are more likely to compromise the target. For example, if a particular sequence of bytes has information about the length of a string that is contained in the next sequence of bytes, a fuzzer can try to increase, decrease or set the length value to a negative number. The target software may not check one of these cases and pass the malformed input to the application logic, resulting in a potentially exploitable memory corruption [14].

2.1 Fuzzing Techniques

There are two methods for creating fuzzed inputs: generation-based fuzzing and mutation fuzzing. To simplify the presentation, we focus on fuzzing packets sent to networked software. The techniques, however, apply generally to fuzz-testing (e.g., of files and file systems).

- **Generation-Based Fuzzing:** This method constructs fuzzed inputs based on generation rules related to valid input structures and protocol states. The simplest generation-based fuzzers generate fuzzed inputs corresponding to random-length strings containing random bytes [15]. State-of-the-art generation-based fuzzers such as Sulley [3] and Peach [11] are typically block-based fuzzers. Block-based fuzzers require a complete description of the input structure in order to generate inputs, and often accept a protocol description as well. SPIKE [1] was the first block-based fuzzer to be distributed. Newer generation-based fuzzers such as EXE [7] instrument code to automatically generate test cases that have a high probability of success.

- **Mutation Fuzzing:** This method modifies good inputs by inserting bad bytes and/or swapping bytes to create fuzzed inputs. Some modern mutation fuzzers base their fuzzing decisions on a description of the input layout (e.g., the mutation aspect of Peach [11]). Other mutation fuzzers such as the General Purpose Fuzzer (GPF) [22] do not require any knowledge of the input layout or protocol; they use simple heuristics to guess field boundaries and accordingly mutate the input. Kaminsky's experimental CFG9000 fuzzer [13] occupies the middle ground by using an adaptation of the Sequitur algorithm [18] to derive an approximation (context-free grammar) of the generative model of a protocol from a sufficiently large traffic capture, and then uses the model to generate mutated inputs. Most mutation fuzzers use previously-recorded network traffic as the basis for mutation, although there are some inline fuzzers that read live traffic. One of the most influential academic works on fuzzing is PROTOS [21], which analyzes a protocol, creates a model and generates fuzzing tests based on the model.

A fuzzing test is typically deemed to be successful when it reveals a bug that harbors a vulnerability. However, in the case of critical infrastructure assets, a broader definition of success is appropriate – discovering a bug that creates any sort of disruption. This is important because any disruption – whether or not it is a security vulnerability – can severely impact the critical infrastructure.

2.2 Inline Fuzzing

In general, most block-based and mutation packet fuzzers work on servers, not clients. This is because these fuzzers are designed to generate packets and send them to a particular IP address and port. Since clients do not accept

traffic that they are not expecting, only fuzzers that operate on live traffic are capable of fuzzing clients. Similarly, protocols that operate in short or time-sensitive sessions are relatively immune to fuzzing that requires a large sample packet dump. For these reasons, inline fuzzing is typically required to fuzz clients.

Fuzzers that are capable of inline fuzzing (e.g., QueMod [12]) either transmit random data or make random mutations. To our knowledge, LZFuzz, which is described in this paper, is the first inline fuzzer that goes beyond random strings and mutations.

2.3 Network-Based Fuzzing

Most modern fuzzers integrate with debuggers to instrument and monitor their targets for crashes. However, using a debugger requires intimate access to the target. Such access is unlikely to be available in the case of most SCADA systems used in the critical infrastructure.

Inline fuzzers like LZFuzz must recognize when the target crashes or becomes unresponsive without direct instrumentation. With some targets, this recognition must trigger a way to (externally) reset the target; other targets may be restarted by hardware or software watchdogs. Note that generation-based fuzzers, which for various reasons cannot leverage target instrumentation, encounter similar challenges. For example, 802.11 Link Layer fuzzers that target kernel drivers [6] have had to work around their successes that caused kernel panics on targets.

In either case, stopping and then restarting the fuzzing iteration over the input space is necessary so that fuzzing payloads are not wasted on an unresponsive target. It is also important for a fuzzer to adapt to its target, especially when dealing with proprietary protocols.

2.4 Fuzzing Proprietary Protocols

It is generally believed that if a fuzzer can understand and adapt to its target, it will be more successful than a fuzzer that does not. Therefore, it is important for a fuzzer to leverage all the available knowledge about the target. When no protocol specifications are available, an attempt can be made to reverse engineer the protocol manually or with the help of debuggers. In practice, this can be extremely time-consuming. Furthermore, it is not always possible to install debuggers on some equipment, which makes reverse engineering even more difficult.

Consequently, it is important to build a fuzzing tool that can work efficiently on proprietary devices and software without any knowledge of the protocol it is fuzzing. Although a mutation fuzzer does not require knowledge of the protocol, it is useful to build a more efficient mutation fuzzer by incorporating field parsing (and other) heuristics that would enable it to respond to protocol state changes on the fly without protocol knowledge. Because instrumenting a target is difficult or impossible in a SCADA environment, the only option is

to employ inline adaptive fuzzing. We refer to this approach as live adaptive mutation fuzzing.

2.5 Fuzzing in Industrial Settings

Proprietary protocols used by SCADA equipment, such as Harris-5000 and Conitel-2020, are often not well-understood. Understandably, domain experts neither have the time nor the skills to reverse engineer the protocols. Fuzzing experts can perform this task, but infrastructure asset owners and operators may be reluctant to grant access to outsiders. In our own experience with power industry partners, it was extremely difficult to gain approval to work with their equipment. Moreover, asset owners and operators are understandably disinclined to share information about proprietary protocols and equipment, making it difficult for outside security experts to perform tests. Clearly, critical infrastructure asset owners and operators would benefit from an effective fuzzing tool that they could use on their own equipment. Our work with LZFuzz seeks to make this possible.

2.6 Modern Fuzzers

This section briefly describes examples of advanced fuzzers that are popular in the fuzz-testing community. Also, it highlights some tools that are available for fuzzing SCADA protocols.

- **General Network-Based Fuzzing Tools:** Sulley [3] is a block-based generation fuzzing tool for network protocols. It provides mechanisms for tracking the fuzzing process and performing *post mortem* analysis. It does so by running code that monitors network traffic and the status of the target (via a debugger). Sulley requires a description of the block layout of a packet in order to generate fuzzed inputs. It also requires a protocol description, which it uses to iterate through different protocol states during the fuzzing process.

  The General Purpose Fuzzer (GPF) [22] is a popular network protocol mutation fuzzer that requires little to no knowledge of a protocol. Although GPF is no longer being maintained, it is one of the few open source modern mutation fuzzers that is commonly available. GPF reads network captures and heuristically parses packets into tokens. Its heuristics can be extended to improve the accuracy with which it handles a protocol, but, by default, GPF attempts to tokenize packets using common string delimiters such as " " and "\n." GPF also provides an interface to load user defined functions that perform operations on packets post-fuzzing.

  Peach is a general fuzzing platform that performs mutation and block-based generation fuzzing [11]. Like Sulley, it requires a description of the fields and protocol. When performing mutation fuzzing, Peach reads in network captures and uses the field descriptions to parse and analyze packets for fuzzing as well as to adjust packet checksums before trans-

mitting the fuzzed packets. Like Sulley, Peach also uses debuggers and monitors to determine success and facilitate *post mortem* analysis.

- **SCADA Fuzzing Tools:** Some tools are available for fuzzing non-proprietary SCADA protocols. In 2007, ICCP, Modbus and DNP3 fuzzing modules were released for Sulley by Devarajan [9]. SecuriTeam includes DNP3 support with its beSTORM fuzzer [4]. Digital Bond created ICCPSic [10], a commercial suite of ICCP testing tools (unfortunately, this suite is no longer publicly available). Also Mu Dynamics offers Mu Test Suite [17], which supports modules for fuzzing SCADA protocols such as IEC61850, Modbus and DNP3.

3. LZFuzz Tool

LZFuzz employs a simple tokenizing technique adapted from the Lempel-Ziv compression algorithm [23] to estimate the recurring structural units of packets; interested readers are referred to [5] for an analysis of the accuracy of the tokenizing method. Effective inputs for fuzzing can be generated by combining this simple tokenizing technique with a mutation fuzzer. The need to understand and model protocol behavior can be avoided by adapting to and mutating live traffic.

In our experience, SCADA protocols used in power control systems perform elaborate initial handshakes and send continuous keep-alive messages. If a target process is crashed, the process will often automatically restart itself and initiate a new handshake.

This behavior is unusual for other non-SCADA classes of targets that need to be specifically instrumented to ensure liveliness and be restarted remotely. Such restarting/session renegotiation behavior assists the construction of successful fuzz sessions. Based on this observation, we propose the novel approach of "adaptive live mutation fuzzing." The resulting fuzzer can adapt its fuzzing method based on the traffic it receives, automatically backing off when it thinks it is successful.

3.1 Design

The LZFuzz tool is inserted into a live stream of traffic, capturing packets sent to and from a source and target. A packet read into LZFuzz gets processed in several steps before it is sent to the target (Figure 1). When LZFuzz receives traffic destined for the target, it first tags the traffic with its type. Then, it applies a set of rules to see if it can declare success. Next, it looks up the LZ string table for the traffic type it is processing, updates the table and parses the packet accordingly. Next, it sends one or more tokens to a mutation fuzzer. Finally, it reassembles the packet, fixing any fields as needed in the packet finishing module. As LZFuzz receives traffic destined for the source, it checks for success and fixes any fields as required before sending the packet to the source.

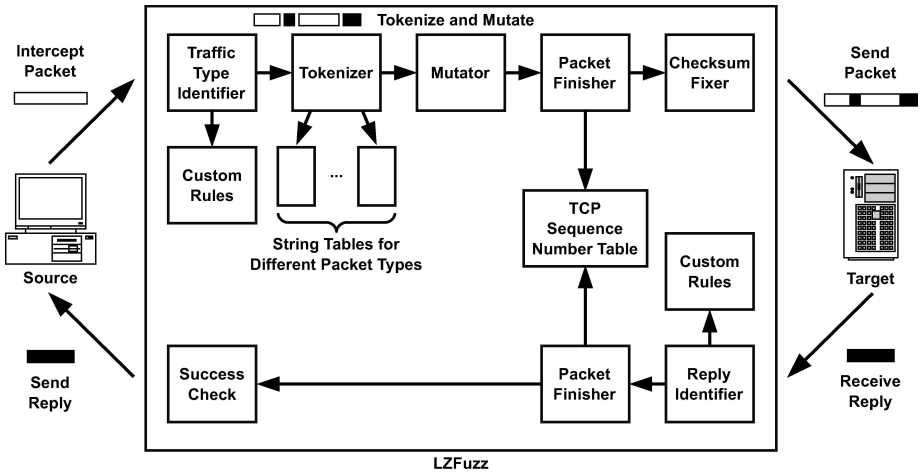

Figure 1. LZFuzz packet processing.

Intercepting Packets. Although it may be possible to configure the source and target to communicate directly with the machine running LZFuzz, it may not always be practical to do so. Consequently, LZFuzz uses a technique known as ARP spoofing or ARP poisoning to transparently insert itself between two communicating parties. This method works when the systems are communicating over Ethernet and IP and at least one of them is on the same LAN switch as the machine running LZFuzz. (In the case of only one target host being local and the remote host located beyond the local LAN, the LAN's gateway must be "poisoned.") The ability to perform ARP spoofing means that fuzzing can be performed without the need to make any direct changes to the source or target configurations. LZFuzz uses the `arp-sk` tool [20] to perform ARP spoofing.

Note that, although various Link Layer security measures exist against ARP poisoning and similar LAN-based attacks can be deployed either at the LAN switches or on the respective hosts or gateways (see, e.g., [8]), such measures are not typically used in control networks, because of the configuration overhead. This overhead can be especially costly in emergency scenarios where faulty or compromised equipment must be quickly replaced, because it is desirable in such situations that the replacement work "out of the box."

Estimating Packet Structure. As LZFuzz reads in valid packets, it builds a string table as if it were performing LZ compression [23]. The LZ table keeps track of the longest unique subsequences of bytes found in the stream of packets. LZFuzz updates its LZ tables for each packet it reads. A packet is then tokenized based on strings found in the LZ table, and each token is treated as if it were a field in the packet. One or more tokens are then passed to GPF, which guesses the token types and mutates the tokens. The number of tokens

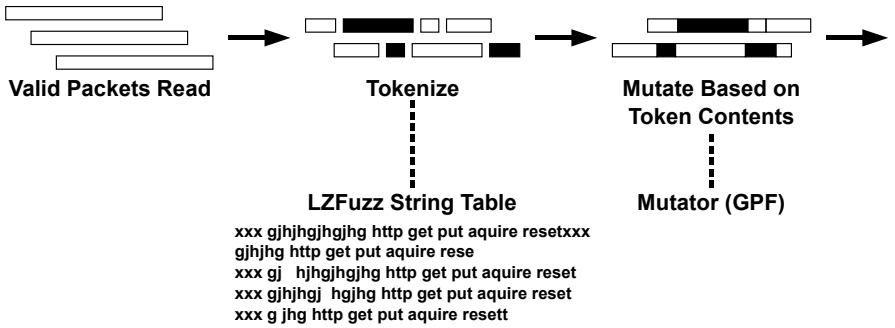

Figure 2. Tokenizing and mutating packets (adapted from [5]).

passed to GPF is dependent on whether or not the windowing mode is enabled. When the mode is enabled, LZFuzz fuzzes one token at a time, periodically changing the token it fuzzes (LZFuzz may fuzz multiple tokens at a time in the windowing mode to ensure that there are enough bytes available to mutate effectively). When the windowing mode is disabled, all the tokens are passed to GPF. Figure 2 provides a high-level view of the tokenizing process.

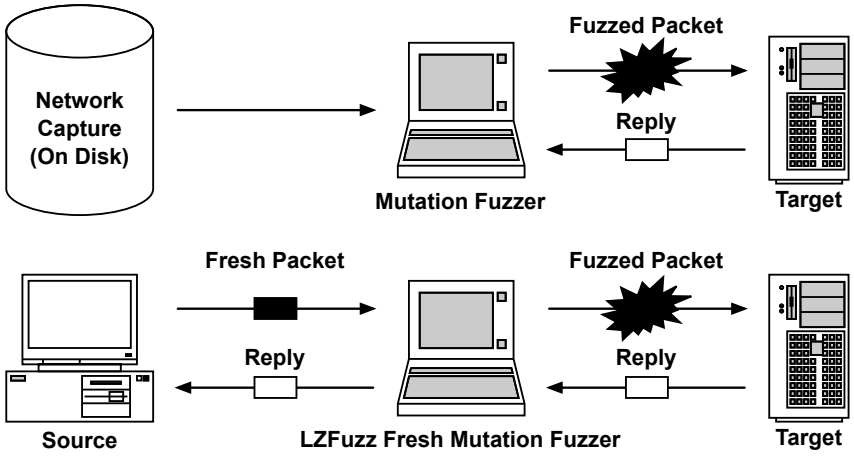

Figure 3. Comparison of live inline mutation with traditional mutation.

Responding to Protocol State Changes. Unlike traditional mutation fuzzers, LZFuzz's mutation fuzzer performs live mutation fuzzing. This means that, instead of mutating previously recorded packets, packets are mutated while they are in transit from the source to the target. Figure 3 shows how live mutation differs from traditional mutation. In particular, live inline mutation enables the fuzzing of short or time-sensitive sessions on real systems in both directions. Traditional mutation fuzzers mutate uncorrupted input

from a network dump whereas LZFuzz mutates packets freshly from a source as it communicates with the target.

Recognizing Target Crashes. Modern network protocol fuzzers tend to require the attachment of a debugger to the target to determine when crashes occur. However, such access is typically not available in SCADA environments. Since live communications are captured as they enter and leave the fuzzing target, our novel approach can make fuzzing decisions based on the types of messages (or lack thereof) sent by the target or source.

SCADA protocols tend to have continuous liveliness checks. If a piece of equipment revives itself after being perceived as dead, an elaborate handshake is typically performed as it reintroduces itself. LZFuzz possesses the ability to recognize such behavior throughout a fuzzing session.

Even if a protocol does not have these keep-alive/handshake properties, other methods can be used to deduce success from network traffic. If a protocol is running over TCP, the occurrence of an RST flag may signify that the target process has crashed. This flag is set when a host receives traffic when it has no socket listening for the traffic. Our experience with LZFuzz has shown that TCP RST flags are a reasonable success metric although they produce some false positives.

Mutation. LZFuzz can work with a variety of fuzzers to mangle the input it fetches. Also, LZFuzz can be easily modified to wrap itself around new mutation fuzzers. Currently, LZFuzz passes packet tokens to the GPF mutation fuzzer for fuzzing before it reassembles the packet and fixes any fields such as checksums.

3.2 Extending LZFuzz

LZFuzz provides an API to allow users to encode knowledge of the protocol being fuzzed. The API can be used to tag different types of packets using regular expressions. New LZ string tables are automatically generated for each type of packet that it is passed. The API also allows users to provide information on how to fix packets before they are sent so that the length and checksum fields can be set appropriately. Finally, the API allows users to custom-define success conditions. For example, if a user knows that the source will attempt a handshake with the target when the target dies, then the user can use the API to tag the handshake packets separately from the data and control packets and to instruct LZFuzz to presume success upon receiving the handshake packets.

4. Experimental Results

An earlier version of LZFuzz was tested on several non-SCADA network protocols, including the iTunes music sharing protocol (DAAP). LZFuzz was able to consistently hang the iTunes version 2.6 client by fuzzing DAAP. It

was also able to crash an older version of the Gaim client by intercepting and fuzzing AOL Instant Messenger traffic.

We selected these protocols because we wanted to test the fuzzer on examples of relatively complex (and popular) client-server protocols that are used for frequent, recurring transactions with an authentication phase separate from the normal data communication phase. Also, we sought protocols that supported some notion of timed and timed-out sessions. Of course, it was desirable that the target software be widely used so that most of the easy-to-find bugs would have presumably been fixed. More importantly, however, LZFuzz was able to consistently crash SCADA equipment used by an electric power company.

Beyond listing successes, it is not obvious how the effectiveness of a fuzzer can be quantitatively evaluated or compared. In practice, a fuzzer is useful if it can crash targets in a reasonable amount of time. But how does one encode such a goal in a metric that can be evaluated?

The best method would be to test the ability of a fuzzer to trigger all the bugs in a target. However, such a metric is flawed because it requires *a priori* knowledge of all the bugs that exist in the target. A more reasonable metric is code coverage – the portion of code in a target that is executed in response to fuzzed inputs. This metric also has its flaws, but it is something that can be measured (given access to the source code of the target), and still provides insight on ability of the fuzzer to reach hidden vulnerabilities. Indeed, in 2007, Miller and Peterson [16] used code coverage to compare generational fuzzing against mutation fuzzing. Also, the usefulness of coverage instrumentation has long been recognized by the reverse engineering and exploit development communities. For example, Amini's PaiMei fuzzing and reverse engineering framework [2] provides the means to evaluate the code coverage of a process up to the basic block granularity; the framework also includes tools for visualizing coverage. Unfortunately, the code coverage metric glosses over differences between a fuzzer constrained to having canned packet traces and one that can operate in a live inline mode. Nevertheless, to provide a means for comparing LZFuzz with other methods of fuzzing proprietary protocols, we set up experiments to compare the code coverage of LZFuzz, GPF and random mutation fuzzing (with random strings of random lengths).

4.1 Experimental Setup

We tested GPF, LZFuzz, random mutation fuzzing and no fuzzing on two targets: `mt-daapd` and the Net-SNMP `snmpd` server. We chose these two targets because `mt-daapd` is an open source server that uses a (reverse engineered) proprietary protocol and Net-SNMP uses the open SNMP protocol used in SCADA systems.

The experiments were conducted on a machine with a 3.2 GHz i5 dual-core processor and 8 GB RAM running Linux kernel 2.6.35-23. Each fuzzing session was run separately and sequentially. The code coverage of the target was measured using `gcov`. The targets were executed within a monitoring environ-

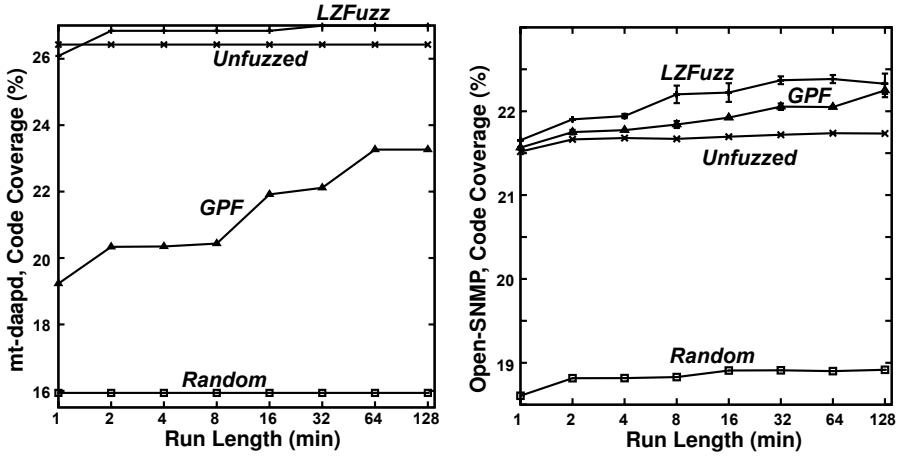

Figure 4. Code coverage for `mt-daapd` (left) and Open-SNMP (right).

ment that would immediately restart the target when a crash was detected (to simulate the automatic reset of common power SCADA applications).

Eight separate tests were conducted on each target; specifically, the fuzzer was run for 1, 2, 4, 8, 16, 32, 64 and 128 minutes. After each test run, the code coverage was computed before resetting the code coverage count for the subsequent run. No fuzzer was provided any protocol information beyond the IP address of the target, the transport layer protocol and the port used by the target. Because GPF uses a network capture as a mutation source, it was supplied with a packet capture of about 1,600 packets as produced by the source/target setup when no fuzzer was active.

4.2 Fuzzing mt-daapd

`mt-daapd` is an open source music server that uses the proprietary iTunes DAA protocol to stream music. This protocol was reverse engineered by several developers who intended to build open source `daapd` servers and clients. We chose `mt-daapd` because we wanted to test a proprietary protocol but required source code in order to calculate code coverage. The tests used `mt-daapd` version 0.2.4.2. The `mt-daapd` daemon was run on the same machine as the client and fuzzer. The server was configured to prevent stray users from connecting to it. The Banshee media player was used as a client and traffic source. To maintain consistency between tests, a set of `xmacro` scripts were used to control Banshee and cause it to send requests to the `daapd` server.

In general, we discovered that, with respect to code coverage, LZFuzz does as well or better than running the test environment without any fuzzer (left-hand side of Figure 4). Furthermore, we found that LZFuzz triggered the largest amount of code in the target compared with the other fuzzers we tested. This means that LZFuzz was able to reach into branches of code that none of the

other fuzzers reached. Interestingly, the random fuzzer consistently produced the same low code coverage on every test run regardless of the length of the run. Other than LZFuzz, no fuzzer achieved higher code coverage than that of a non-fuzzed run of Banshee and `mt-daapd`.

4.3 Fuzzing snmpd

Net-SNMP is one of the few open source projects that use SCADA protocols. Our experiments used `snmpd`, the daemon that responds to SNMP requests in Net-SNMP version 5.6.1, as a fuzzing target. Like `mt-daapd`, the daemon was run on the same system as the client. We scripted `snmpwalk`, provided by Net-SNMP, to continuously send queries to the server. For the purpose of code coverage testing, `snmpwalk` was used to query the status of several parameters, including the system uptime and location, and information about open TCP connections on the system. Because we were unable to make consistent code coverage measurements between runs of the same fuzzer and run length, we ran each fuzzer and run length combination five times. The averages are displayed in Figure 4 (right-hand side) along with error bars for runs with noticeable variation (standard deviation greater than 0.025%).

GPF outperformed LZFuzz when GPF was running at full strength. However, we were also interested in seeing the relative performance of LZFuzz and GPF when GPF had a rate-adjusted flow so that GPF would send about the same number of packets as LZFuzz for a given run length. This adjustment provided insight into how much influence a GPF-mutated packet would have on the target compared with a LZFuzz-mutated packet. We also observed that LZFuzz induced a larger amount of code coverage in `snmpd` when the mutation rate that controlled the mutation engine aggressiveness was set to medium (instead of high or low). The mutation rate governs how much the GPF mutation engine mutates a packet. Although this feature is not documented, the mutation rate is required to be explicitly set during a fuzzing session. Line 143 of the GPF source file `misc.c` offers the options "high," "med" or "low" without any documentation; we chose the "med" option for `snmpd` and "high' for `mt-daapd`. Because GPF uses the same mutation engine, we set GPF to run with the medium mutation level as well. Note that, in the case of `snmpd`, a 1% difference in code coverage corresponds to about 65 lines of code.

Figure 4 (right-hand side) shows the code coverage of GPF (with a rate-adjusted flow and a medium mutation rate) compared with the code coverage of LZFuzz (with medium mutation), random fuzzing and no fuzzing. With rate-adjusted flow, LZFuzz induces a higher code coverage than GPF. LZFuzz also clearly outperforms random fuzzing.

Although LZFuzz and GPF share a common heuristic mutation engine, they belong to different classes of fuzzers and each has its own strengths and weaknesses. LZFuzz can fuzz both servers and clients; GPF can only fuzz targets that are listening for incoming traffic on a port known to GPF before the fuzzing session. LZFuzz is restricted to only fuzzing packets sent by the source; GPF can send many packets in rapid succession; GPF requires the user to prepare a

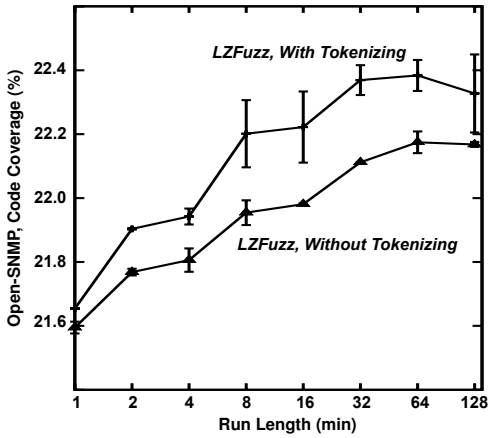

Figure 5. Code coverage for the Net-SNMP server with and without tokenizing.

representative packet capture and, thus, implicitly assumes that representative captures exist for the target scenario. Note that the time taken to prepare the network capture was not considered in our results.

The packet capture given to GPF potentially provided it with a wealth of information about the protocol from the very beginning. On the other hand, LZFuzz had to develop most of its knowledge about the protocol on the fly. Also, the mutation engine of GPF was built and tuned specifically for what GPF does – fuzzing streams of packets. LZFuzz uses the same mutation engine, but only had one packet in each stream. While the GPF mutation engine was not designed to be used in this manner, we believe that the effectiveness of LZFuzz can be improved if the mutation engine could be tuned.

When GPF and LZFuzz were used at full strength against `mt-daap`, LZFuzz outperformed GPF in terms of code coverage. However, this was not the case when both fuzzers were tested against `snmpd` – GPF achieved 1–2% more code coverage than LZFuzz in comparable runs. It could be argued that GPF is the more effective fuzzer for `snmpd`. However, the clear advantage of LZFuzz over GPF and other similar fuzzers is that it can also fuzz SNMP clients (e.g., `snmpwalk`) whereas GPF cannot do this without session-tracking modifications.

4.4 LZFuzz Tokenizing

The final issue to examine is whether or not the LZFuzz tokenizing method improves the overall effectiveness of the tool. If tokenizing is disabled in LZFuzz during a run and the entire payload is passed directly to GPF, then GPF attempts to apply its own heuristics to parse the packet. Figure 5 shows how LZFuzz with tokenizing compares with LZFuzz without tokenizing when run against `snmpd` in the same experimental environment as described above. These results suggest that the LZ tokenizing does indeed improve the effectiveness of inline fuzzing with the GPF mutation engine.

5. Conclusions

The LZFuzz tool enables control systems personnel with limited fuzzing expertise to effectively fuzz proprietary protocol implementations, including the SCADA protocols used in the electric power grid. LZFuzz's adaptive live mutation fuzzing approach can fuzz the proprietary DAA protocol more efficiently than other methods. LZFuzz is also more effective than applying a random fuzzer to an SNMP server. The GPF mutation fuzzer appears to be more effective at fuzzing an SNMP server than LZFuzz; however, unlike LZFuzz, GPF is unable to fuzz SNMP clients.

Additional work remains to be done on LZFuzz to ensure its wider application in the critical infrastructure. The user interface must be refined to change the aggressiveness of fuzzing or temporarily disable fuzzing without having to restart LZFuzz. Another refinement is to identify checksums by intercepting traffic to the target and passively search for bytes that appear to have high entropy. Also, the tool could be augmented to test for authentication and connection setup traffic by inspecting traffic at the beginning of a run and traffic from the target after blocking replies from the client, and vice versa. This information can be used to specify traffic rules that would make LZFuzz more effective.

Note that the views and opinions in this paper are those of the authors and do not necessarily reflect those of the United States Government or any agency thereof.

Acknowledgements

This research was supported by the Department of Energy under Award No. DE-OE0000097. The authors wish to thank Axel Hansen and Anna Shubina for their assistance in developing the initial prototype of LZFuzz. The authors also wish to thank the power industry personnel who supported the testing of LZFuzz in an isolated environment at their facility.

References

[1] D. Aitel, An introduction to SPIKE, The fuzzer creation kit, presented at the *BlackHat USA Conference* (www.blackhat.com/presentations/bh-usa-02/bh-us-02-aitel-spike.ppt), 2002.

[2] P. Amini, PaiMei and the five finger exploding palm RE techniques, presented at *REcon* (www.recon.cx/en/s/pamini.html), 2006.

[3] P. Amini, Sulley: Pure Python fully automated and unattended fuzzing framework (code.google.com/p/sulley), 2010.

[4] Beyond Security, Black box software testing, McLean, Virginia (www.beyondsecurity.com/black-box-testing.html).

[5] S. Bratus, A. Hansen and A. Shubina, LZFuzz: A Fast Compression-Based Fuzzer for Poorly Documented Protocols, Technical Report TR2008-634, Department of Computer Science, Dartmouth College, Hanover, New Hampshire (www.cs.dartmouth.edu/reports/TR2008-634.pdf), 2008.

[6] J. Cache, H. Moore and M. Miller, Exploiting 802.11 wireless driver vulnerabilities on Windows, *Uninformed*, vol. 6 (uninformed.org/index.cgi?v=6), January 2007.

[7] C. Cadar, V. Ganesh, P. Pawlowski, D. Dill and D. Engler, EXE: Automatically generating inputs of death, *ACM Transactions on Information and System Security*, vol. 12(2), pp. 10:1–38, 2008.

[8] S. Convery, Hacking Layer 2: Fun with Ethernet switches, presented at the *BlackHat USA Conference* (www.blackhat.com/presentations/bh-usa-02/bh-us-02-convery-switches.pdf), 2002.

[9] G. Devarajan, Unraveling SCADA protocols: Using Sulley fuzzer, presented at the *DefCon 15 Hacking Conference*, 2007.

[10] Digital Bond, ICCPSic assessment tool set released, Sunrise, Florida (www.digitalbond.com/2007/08/28/iccpsic-assessment-tool-set-released), 2007.

[11] M. Eddington, Peach Fuzzing Platform (peachfuzzer.com), 2010.

[12] GitHub, QueMod, San Francisco (github.com/struct/QueMod), 2010.

[13] D. Kaminsky, Black ops: Pattern recognition, presented at the *BlackHat USA Conference* (www.slideshare.net/dakami/dmk-blackops2006), 2006.

[14] H. Meer, Memory corruption attacks: The (almost) complete history, presented at the *BlackHat USA Conference* (media.blackhat.com/bh-us-10/white papers/Meer/BlackHat-USA-2010-Meer-History-of-Memory-Corruption-Attacks-wp.pdf), 2010.

[15] B. Miller, L. Fredriksen and B. So, An empirical study of the reliability of UNIX utilities, *Communications of the ACM*, vol. 33(12), pp. 32–44, 1990.

[16] C. Miller and Z. Peterson, Analysis of Mutation and Generation-Based Fuzzing, White Paper, Independent Security Evaluators, Baltimore, Maryland (securityevaluators.com/files/papers/analysisfuzzing.pdf), 2007.

[17] Mu Dynamics, Mu Test Suite, Sunnyvale, California (www.mudynamics.com/products/mu-test-suite.html).

[18] C. Nevill-Manning and I. Witten, Identifying hierarchical structure in sequences: A linear-time algorithm, *Journal of Artificial Intelligence Research*, vol. 7, pp. 67–82, 1997.

[19] T. Proell, Fuzzing proprietary protocols: A practical approach, presented at the *Security Education Conference Toronto* (www.sector.ca/presentations10/ThomasProell.pdf), 2010.

[20] F. Raynal, E. Detoisien and C. Blancher, `arp-sk`: A Swiss knife tool for ARP (sid.rstack.org/arp-sk), 2004.

[21] J. Roning, M. Laakso, A. Takanen and R. Kaksonen, PROTOS: Systematic approach to eliminate software vulnerabilities, presented at Microsoft Research, Seattle, Washington (www.ee.oulu.fi/research/ouspg/PROTOS_MSR2002-protos), 2002.

[22] VDA Labs, General Purpose Fuzzer, Rockford, Michigan (www.vdalabs.com/tools/efs_gpf.html), 2007.

[23] J. Ziv and A. Lempel, A universal algorithm for sequential data compression, *IEEE Transactions on Information Theory*, vol. 23(3), pp. 337–343, 1977.