Chapter 1

DO-IT-YOURSELF SCADA VULNERABILITY TESTING WITH LZFUZZ

Rebecca Shapiro, Sergey Bratus, Edmond Rogers and Sean Smith

Abstract Security vulnerabilities typically start with bugs: in input validation, and also in deeper 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 such bugs. However, for SCADA software used in critical infrastructure, the widespread use of proprietary protocols makes it difficult to apply existing fuzz-testing techniques, which work best when protocol semantics are known, targets can be instrumented, or at least large network traces are available. These things typically don't apply in real-world infrastructure such as power SCADA. Domain experts often do not have the time and data to understand the proprietary protocols their equipment uses well enough for fuzz-testing. Domain experts are understandably unwilling to share sufficient internal access to allow external security experts to perform the fuzz-testing; and the domain uses live sessions with short data validity time window, which makes it hard to prime a fuzzer with large network capture dumps.

This paper presents a solution: LZFuzz, a man-in-the-middle, *inline* fuzz-testing appliance which provides a domain expert with tools to effectively fuzz SCADA equipment.

Keywords: Vulnerability assessment, SCADA

1. Introduction

Critical infrastructure such as the power grid is monitored and controlled by *supervisory control and data acquisition (SCADA)* systems. Proper functioning of these systems is necessary to ensure the stability of such infrastructure; something as simple as an input validation bug in SCADA software can leave our infrastructure vulnerable to attack. While large software development companies may have the resources to develop techniques and tools to thoroughly test their software, our experience has shown that the same cannot be said for manufactures of SCADA equipment. Dr. Thomas Pröll from Siemens has made similar observations, finding that random streams of bytes are often enough to crash SCADA equipment [18].

Securing SCADA systems requires testing for such vulnerabilities. However, we have a Catch-22 situation here:

- Software vulnerabilities are often not well understood by developers and infrastructure experts (who may themselves not even have full protocol documentation).

- External security experts lack the domain knowledge, resources, and access to run thorough tests.

Fuzz-testing, a form of security testing in which bad inputs are chosen in attempt to crash the software, is a widely used method of testing for security bugs, both in input validation and also in deeper application logic. However, using fuzz-testing methodologies to secure such SCADA systems is difficult. SCADA equipment often relies on poorly understood proprietary protocols, complicating test development. The session-oriented, short or time-sensitive (session data remains valid for a short time only, and will likely get rejected out of hand by the target) nature of many SCADA scenarios (such as those we've worked with in the power grid) make it impossible to prime a fuzzer with a large capture; we need something that works inline. Furthermore, many modern fuzzers require users to attach a debugger to the target — this is not always possible in our target scenarios.

This paper presents a solution: LZFuzz, a fuzzing appliance designed to allow asset owners to effectively fuzz their own equipment without needing to modify the target system being tested — and without needing to expose assets and information to external security evaluators. Section 2 provides an overview of fuzzing. Section 3 presents our project. Section 4 presents experimental results. Section 5 concludes.

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 [13]. The addition of randomness to the input triggered bugs that were never found during the software's testing processes. After further investigation, Miller found that the types of bugs that were triggered from the fuzzing included race conditions, buffer overflows, failure to check return code and printf/format string problems. Such types of bugs are often sources of security vulnerabilities in software

[12]. Today, most modern software have aggressive input checking and can handle random streams of bytes without crashing. Consequently, modern fuzz-testing tools have adapted to be more effective on such modern software, by becoming more selective of how they fuzz inputs.

There usually are multiple layers of processing that data has to go through before it reaches the target software's application logic whether or not the data has been fuzzed. We can think of a target's application logic as its soft underbelly — if we can penetrate it we can greatly increase our chances of compromising the target. Fuzzed input has the ability to trigger bugs or to be rejected by any layer of processing it passes through before it gets to the application logic. To trigger bugs in a target's application logic, the fuzzer needs to generate inputs that are clean enough to pass any sanity checks in the application but malformed enough to trigger bugs in the application logic.

In practice, the most successful fuzzers create fuzzed input based on *full knowledge* of the layout and contents of the input. If a fuzzer is given information on how a specific byte is going to 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 contains information on 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's sanity checks may miss one of these cases and blindly pass the malformed input on to the application logic.

Fuzzing Methods. There are two different approaches to creating fuzzed input: *generation-based fuzzing* and *mutation fuzzing.* In our subsequent discussion of these methods, we will focus on fuzzing packets sent to *networked software* in order to simplify our explanations and descriptions. (These methods, however, apply generally to fuzz-testing.)

Generation-based fuzzers construct their fuzzed input based on generation rules such as how valid input is structured or what the protocol states are. The most simple generation-based fuzzers, such as the one used by Miller in [13], generate fuzzed input by creating strings of random bytes and lengths. Today's state-of-the-art generation-based fuzzers, such as Sulley [21], fall into the subcategory of *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], created by Dave Aitel, was the first block-based fuzzer to be distributed. New generation-based fuzzers, such as EXE [7], instrument code to automatically generate test cases that have a high probability of succeeding.

Mutation fuzzers operate by reading in known good inputs and inserting badness and swapping bytes to create fuzzed inputs. Some modern mutation fuzzers such as the mutation aspect of Peach [17] make their fuzzing decisions based on a description of the input layout. Other mutation fuzzers such as General Purpose Fuzzer (GPF) [9] do not require any knowledge of the protocol or layout. GPF uses simple heuristics to guess field boundaries and mutates the input based on its guesses. Such fuzzers do not need to know anything about the structure of what they are fuzzing. They can blindly read in known good inputs and mutate without knowing the semantics of the protocol they are fuzzing. Kaminsky's experimental CFG9000 fuzzer [11] occupies a middle ground by using an adaptation of the Sequitur algorithm [16] to derive an approximation of the protocol's generative model (specifically, a context-free grammar) from a sufficiently large traffic capture, and then using than model to generate the mutated inputs.

Most mutation fuzzers use previously recorded network traffic as a basis for mutation, although there have been some inline fuzzers that read in live traffic.

The most influential academic work on fuzzing is arguably the PROTOS toolkit [20] developed at the University of Oulu. PROTOS first analyzes a protocol and then generates fuzzing tests based on a model it generated.

Fuzzing Successes. Historically, success is declared when a fuzzer reveals a bug harboring a vulnerability. However, for critical infrastructure, we are considering a broader definition of success: discovering a software bug that creates any sort of disruption. We care about *all* disruptions because any disruption, whether or not it is a security vulnerability, may affect the stability of the system of which it is a component.

2.1 Inline Fuzzing

In general, most block-based and mutation packet fuzzers available are only able to fuzz servers, whereas clients are left in the dark. The reason such fuzzers cannot fuzz clients is because fuzzers are designed to generate packets and send them to a particular IP address and port. Since clients will not receive traffic 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 may not be amenable to fuzzing that requires a large sample packet dump—an inline fuzzer is required.

The only fuzzers we have found that are capable of inline fuzzing, such as QueMod [19], either transmit random data or make random mutations. To our knowledge, LZFuzz is the first inline fuzzer that goes beyond random strings and mutations.

2.2 Network-Based Fuzzing

Most modern fuzzers come packaged with debuggers to instrument and monitor their targets for crashes. However, using such debuggers requires intimate access to the target system; in the power SCADA cases we examined, such access was not possible.

Inline fuzzers such as LZFuzz must recognize when the target crashes or is unresponsiveness without direct instrumentation. With some targets, this recognition must trigger a way to (externally) reset and the target, whereas other targets may be restarted by hardware or software watchdogs. We note that generation-based fuzzers that for various reasons cannot take advantage of target instrumentation encounter similar challenges: for example, 802.11 Link Layer fuzzers targeting kernel drivers [6] had to work around its successes causing kernel panics on the targets.

In either case, stopping and then restarting the fuzzing iteration over the input space is required so that generated fuzzing payloads are not wasted on an already unresponsive target. We believe that it is important for a fuzzer to be able to adapt to its target in these ways, particularly when dealing with proprietary protocols.

2.3 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 no. Therefore, it is important for fuzzers to be able to take advantage of any knowledge available about the target. When no protocol specifications are available, we can work to reverse engineer the protocol manually or with the help of debuggers. In practice, this can be extremely time-consuming and thus is rarely an option for SCADA asset owners. Furthermore, it is not always possible to install debuggers on some equipment — thus making reverse engineering even more difficult.

Consequently, it is important to build fuzzing tools that can work efficiently on all equipment and proprietary software without *any* knowledge of the protocol they are fuzzing. Although mutation fuzzers do not need knowledge of the protocol, we can build a more-efficient generation of mutation fuzzers that have better field-parsing heuristics and that can respond to protocol state changes on the fly without protocol knowledge. Because our SCADA fuzzing goals make target instrumentation difficult or impossible, we are compelled to focus on an inline adaptive fuzzing framework. We refer to this approach as *live adaptive mutation fuzzing.*

2.4 Fuzzing in the Industrial Setting

Proprietary protocols that SCADA equipment use — such as Harris-5000 and Conitel-2020 — are often not well-understood. Understandably, domain experts usually do not have the time or the skills to reverse engineer the protocols their equipment uses. Fuzzing experts can be called in to do the work needed to fuzz the equipment; however in certain domains, such as the power grid, the domain experts are wary of allowing outsiders to work with their specialized equipment. In our own experience with power industry partners, it was extremely difficult to obtain permissions to allow our own researchers work with their equipment. Domain experts are also understandably disinclined to share any information on the proprietary protocols their equipment used, making it difficult for a security expert to perform tests.

Domain experts in critical infrastructure would benefit from an effective fuzzing appliance that they easily use on their own equipment. We seek to fill this gap.

2.5 Modern Fuzzers

In this section we will briefly describe examples of advanced fuzzers that are popular in the fuzz-testing community. We also introduce some tools that are available for fuzzing SCADA protocols.

General Network-Based Fuzzers. *Sulley*, mentioned above, is a block-based generation fuzzing framework for network protocol fuzzing [21]. It provides mechanisms for tracking the process of the fuzzing job and performing postmortem analysis. It does so by running code that monitors both the network traffic and the status of the target (via a debugger). Sulley requires a description of the block layout in a packet in order to generate fuzzed inputs. It also requires a protocol description which it uses to iterate through different protocol states as it fuzzes.

The *General Purpose Fuzzer* (GPF) is a popular network protocol mutation fuzzer that requires little to no knowledge of the protocol it is operating on [9]. Although GPF is not being maintained anymore, it is one of the few open-source modern mutation fuzzers available. GPF reads in network captures and heuristically parses packets into to tokens to fuzz. The heuristics it uses can be extended to improve accuracy for any protocol, but by default GPF attempts to tokenize packets using common string deliminators 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 framework that performs both mutation and block-based generation fuzzing [17]. Like Sulley, it requires a description of the fields and protocol to operate. When performing mutation fuzzing it reads in network captures and uses the field descriptions to parse and analyze the packet for fuzzing as well as to fix the packet's checksums, etc, before sending. Like Sulley, Peach also uses debuggers and monitors to determine success and in order to aid postmortem analysis.

SCADA Fuzzing Tools. There are a handful of tools available that have been built for fuzzing non-proprietary SCADA protocols. In 2007 Ganesh Devarajan from TippingPoint released DNP3, ICCP, and Modbus fuzzing modules for Sulley [21]. The details for these protocols are well-known and have been published. SecuriTeam includes DNP3 support with their commercial *beSTORM* fuzzer [4]. Digital Bond offers a commercial suite of ICCP testing tools called *ICCPSic* [10].[1] Also Mu Dynamics offers commercial software called *Mu Test Suite* [15] which supports modules for fuzzing standardized SCADA protocols such as IEC61850, MODBUS, and DNP3.

3. The LZFuzz Project

3.1 Our Approach

Often the full details of a proprietary protocol are unknown outside the company in which they were engineered. When we do not know protocol details, the current state of fuzz-testing leaves us with three options: we can either fuzz with random streams of bytes, or randomly mutate live or pre-captured traffic with few heuristics, or we can attempt to reverse-engineer the protocol — but reverse-engineering a protocol takes resources power experts do not have. We believe LZFuzz hits a balance between speed and accuracy while still producing protocols models that are effective for fuzzing.

Our method employs a simple tokenizing technique adapted from the Lempel-Ziv (LZ) compression algorithm [22] to estimate packet structural units. By combining this simple tokenizing technique with a mutation fuzzer we believe we can generate effective inputs for fuzzing. Our preliminary technical report explores the accuracy of the tokenizing method in depth [5]. We can avoid the need to understand and model the protocol's behavior by adapting to and performing mutation on live traffic.

In our experience we have found that SCADA protocols used in power control systems perform elaborate initial handshakes and send continuous keep-alive messages. If we crash the target process, the process will often automatically restart itself and initiate a new handshake.[2]

This behavior is unusual for other classes of targets which need to be specifically instrumented to insure liveliness and be restarted remotely. Such restarting/session renegotiation behavior assists our construction of successful fuzz sessions. From this observation, we propose the novel approach of *adaptive* live mutation fuzzing. Our fuzzer can adapt its fuzzing method based on the traffic it receives, automatically backing off when it thinks it is successful.

3.2 Design

Overview. LZFuzz is an appliance that inserts itself into a live stream of traffic, capturing packets that are being sent to and from a source and target. A packet read into LZFuzz gets processed in several steps before being sent to the target. Figure 1 maps these steps out. When LZFuzz receives traffic destined for the target, it first tags the traffic with its type. Then, it reads through 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 needed before sending the packet to the source.

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 being located beyond the local LAN, the LAN's gateway must be "poisoned".) When we have the ability to perform ARP spoofing we do not have to make any direct changes to the source or the target's configuration in order to perform our fuzzing. The particular tool LZFuzz uses to perform the ARP spoofing is arp-sk [3].

We note that although various Link Layer security measures against ARP poisoning and similar LAN-based attacks can be deployed either at the LAN switches or on the respective hosts or gateways themselves

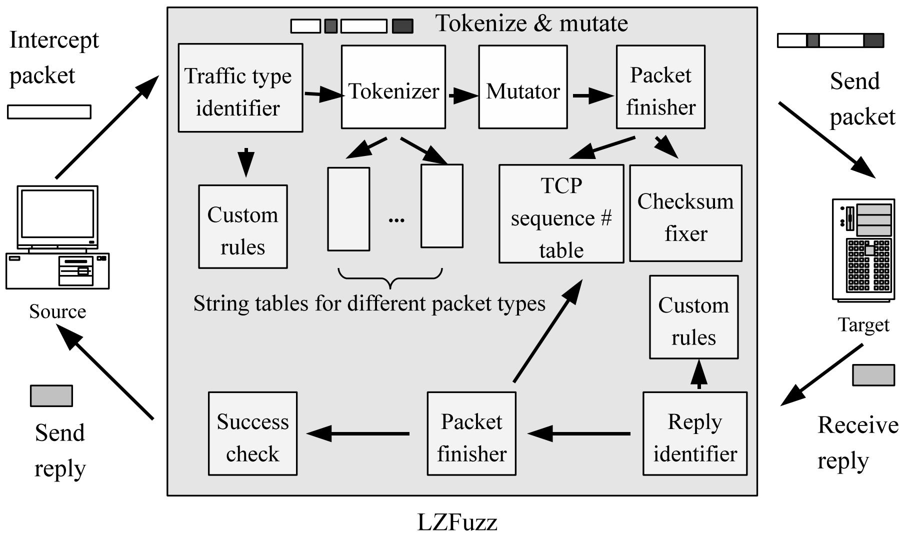

Figure 1. How packets are processed by LZFuzz

(see, e.g., [8]), such measures are not typically used in control networks, because of the configuration overhead they require. This overhead can be especially costly in emergency scenarios where faulty or compromized equipment must be quickly replaced, as it is desirable in such situations that the replacement would work "out of the box".

Estimating Packet Structure. As LZFuzz reads in valid packets it builds a string table as if it were to perform LZ compression [22]. The LZ table keeps track of all 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 is then passed tot GPF where GPF guesses the token types and mutates the tokens. The number of tokens passed to GPF is dependent on whether or not windowing mode is enabled. When enabled, LZFuzz will fuzz one token at a time, periodically changing which token it fuzzes.[3] When windowing mode is disabled, all tokens are passed to GPF.

Figure 2 gives a high-level view of the tokenizing process.

Responding to Changes Protocol State. A major difference between our mutation fuzzer and other existing mutation fuzzers is that

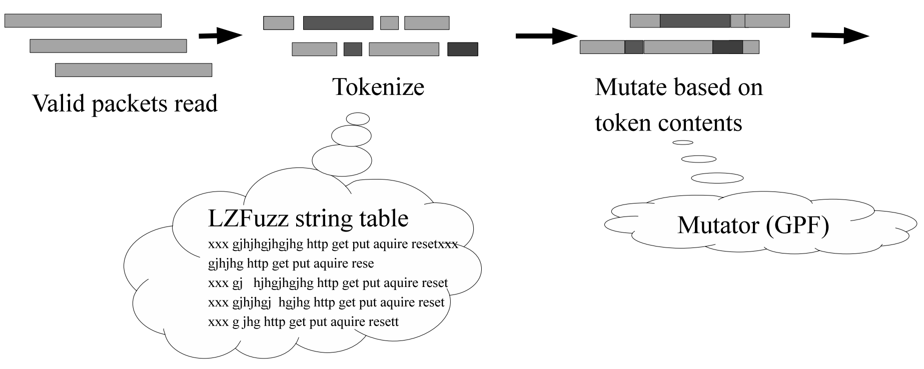
Valid packets read
Tokenize
Mutate based on token contents
LZFuzz string table
xxx gjhjhgjhgjhg http get put aquire resetxxx
gjhjhg http get put aquire rese
xxx gj hjhgjhgjhg http get put aquire reset
xxx gjhjhgj hgjhg http get put aquire reset
xxx g jhg http get put aquire resett
Mutator (GPF)

Figure 2. Tokenizing and mutating packets. Image adapted from our preliminary technical report [5]

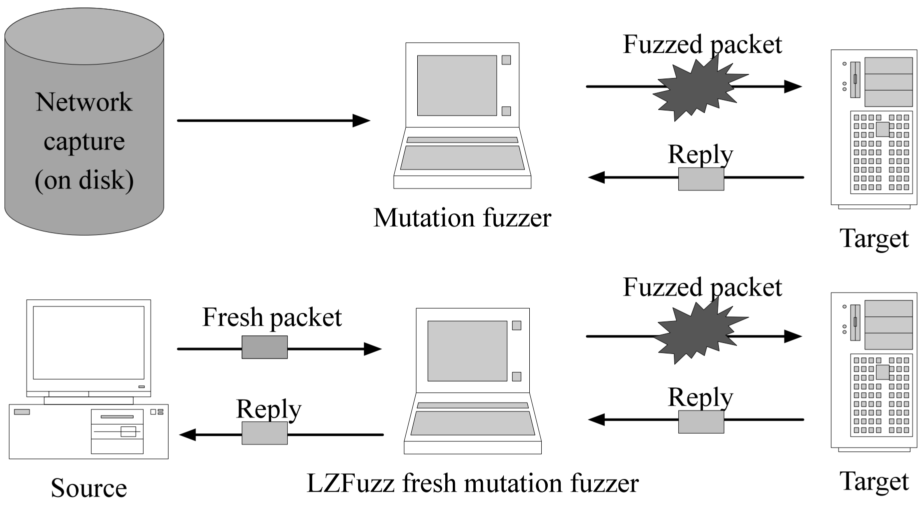

Figure 3. Live, inline mutation enables us to fuzz short or time-sensitive sessions on real systems in both directions.

ours performs *live mutation* fuzzing. This means instead of mutating previously recorded packets, we mutate packets while they are in transit from the source to the target. Figure 3 depicts how the live mutation process differs from the existing mutation process. Other mutation fuzzers mutate uncorrupted input from a network dump whereas LZ-Fuzz mutates packets freshly from a source as it communicates with the target.

Recognizing Target Crashes. Modern network protocol fuzzers require attaching a debugger to the target to determine when crashes happen. However, our target scenarios don't give us that access. Consequently, because we are transparently fuzzing intercepted traffic, we must infer success elsewhere. Because we are capturing live communication as it enters and leaves the fuzzing target, our novel approach can make fuzzing decisions based on what types of messages (or lack thereof) are being sent by the target or source.

In our experience with SCADA protocols, we have observed that these protocols tend to have continuous liveliness checks. If a piece of equipment revives itself after being perceived as dead, there is often an elaborate handshake as it reintroduces itself. LZFuzz has the capability of recognizing such behaviors throughout a fuzzing session.

Even if a protocol doesn't have these keep-alive/handshake properties, there are other methods of deducing success from the 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 will be set when a host receives traffic while it had no socket listening for the traffic. Our experience with LZFuzz has shown us that TCP RST flags appear be a reasonable success metric even though they produce some false positives.

Mutation. LZFuzz has the ability to work with a variety of fuzzers to mangle the input it fetches. It can be easily modified to wrap itself around new state-of-the-art mutation fuzzers. Currently, LZFuzz passes packet tokens to the GPF mutation fuzzer (described in Section 2.5) for fuzzing before it reassembles the packet and fixes any fields such as checksums as needed.

3.3 Extending LZFuzz

The LZFuzz appliance provides an API to allow users to encode any knowledge they have of the protocol being fuzzed. It can also be used to tag different types of packets using regular expressions, for example. LZFuzz will automatically generate new LZ string tables for each type of packet it is passed. The API also allows users to provide information on how to fix packets before sending so that any 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 this API to tag the handshake packets separately from the data and control packets and to instruct our appliance to presume success upon receiving handshake packets.

4. Experimental Results

When LZFuzz was in its infancy, we tested it on some network protocols we had around the lab; we were able to consistently hang the iTunes version 2.6 client by fuzzing the iTunes music sharing protocol (daap). LZFuzz also able to crash an older version of the Gaim client by intercepting and fuzzing AOL Instant Messenger traffic.

We chose to these protocols because we wanted to test our fuzzer on examples of popular, modern, relatively complex client-server protocols that are used for frequent, recurring transactions with an authentication phase separate from the normal data communication phase. Also, we wanted the example protocols to support some notion of time and timed-out sessions. It was preferable that the target software be in common use by many people so that the easier to find bugs have already presumably been fixed. More importantly, however, LZFuzz was able to consistently crash SCADA equipment owned by a power partner.

Beyond listing successes, it is not particularly obvious as to how to quantitatively evaluate or compare the effectiveness of fuzzers. In practice, a fuzzer is useful if it is able to crash targets in a reasonable amount of time. How do we encode such a goal in an actual metric that we can evaluate? The best measure is to test the fuzzer's ability to trigger *all* bugs in a target. However, such a metric is impossible to measure because that would require knowledge of all bugs in an application to begin with. A more reasonable measure would be to calculate *code coverage* — the portion of code in the target that was executed in response to the fuzzed inputs. This metric, too, 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 the fuzzer's ability to reach hidden vulnerabilities. Indeed, in 2007 Miller et al. of Independent Security Evaluators used code coverage as a metric to compare generational fuzzing against mutation fuzzing [14]. Furthermore, the usefulness of coverage instrumentation has has long been recognized by the reverse engineering and exploit development community; for example, Pedram Amini's PaiMei fuzzing and reverse engineering framework provides the means to evaluate code coverage of a process up to the basic block granularity [2]. Interestingly, his framework also includes tools for visualizing coverage. Unfortunately, this metric glosses over differences between a fuzzer constrained to having canned packet traces and one that can work live, inline.

In order to provide a means of comparing our method of fuzzing to other existing methods of fuzzing proprietary protocols, we set up experiments to compare code coverage of LZFuzz, GPF (the mutation fuzzer which LZFuzz itself interfaces with), and fuzzing with random strings of random lengths. (As mentioned previously, Dr. Thomas Pröll was able to crash SCADA equipment with random fuzzing [18].)

4.1 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 seen in SCADA systems.

All experiments we ran were conducted on a machine with a 3.2GHz i5 dual core processor and 8GB RAM running Linux kernel 2.6.35-23. Each fuzzing session was run separately and sequentially. We measured the code coverage of the target using gcov. In all tests, targets were run inside a monitoring environment that would immediately restart the target if a crash was detected to simulate the automatic reset of common power SCADA applications. Each code coverage measurement reported is a product of a separate and isolated run.

For each target, we conducted 8 separate tests, allowing the fuzzer to run for 1, 2, 4, 8, 16, 32, 64, and 128 minutes. After each test run, we calculated code coverage before resetting the code coverage counts for the subsequent run. No fuzzer was provided information about the protocol it was fuzzing beyond the IP address of the target the transport layer protocol and port that the target was using. Because GPF uses a network capture for a mutation source, we supplied GPF with a packet capture of about 1600 packets as produced by the source/target setup when no fuzzer was active.

4.2 mt-daapd

mt-daapd is an open source music server that uses the proprietary iTunes daapd protocol to stream music. This protocol was reverse-engineered by a variety of developers so that they could build open source daapd servers and clients. We choose mt-daapd specifically because we wanted to test a proprietary protocol but required source code in order to calculate code coverage. In our tests, we fuzzed 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. We used the Banshee media player as a client and traffic source. To maintain consistency between tests, we used a set of xmacro scripts to control Banshee and cause it to send requests to the daap server.

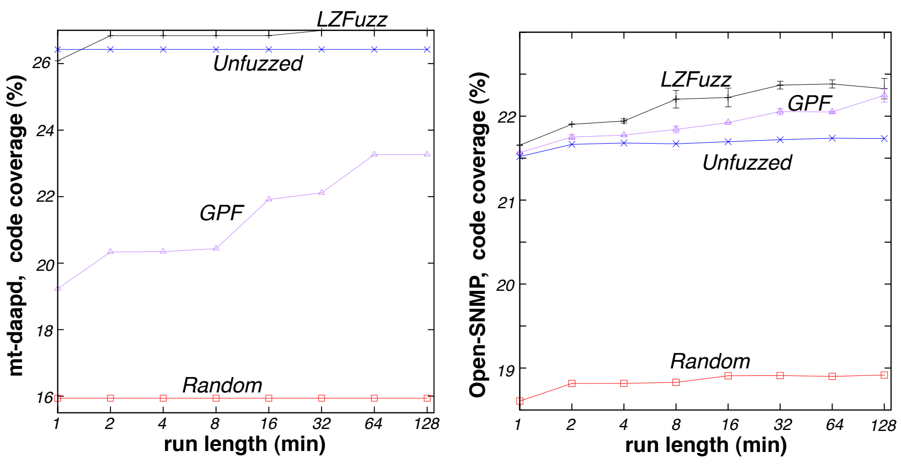

Figure 4. Comparison of code coverage of mt-daapd (left) and Open-SNMP (right) as produced by different fuzzing methods

Results. In general, we found that LZFuzz does as well or even better, in terms of code coverage, than running the test environment without any fuzzer, as shown on the left in Figure 4. Furthermore, we found that of all fuzzers, LZFuzz was able to trigger the largest amount of code in the target. This means that LZFuzz was able to reach into branches of code that none of the other fuzzers we tested were able to reach. It is also worth noting that the random fuzzer consistently achieved the same code coverage on every test run regardless of the length of the run.

Other than LZFuzz, no fuzzer that we tested in this scenario was able to achieve higher code coverage than that of a non-fuzzed run of Banshee and mt-daapd.

4.3 snmpd

Net-SNMP is a suite of open source SNMP *(simple network management protocol)* tools which include snmpd, an SNMP server. SNMP is a flexible monitoring and management protocol used to monitor networks and other systems both inside and outside of SCADA environments. Net-SNMP is one of the few open source projects we found that used SCADA protocols. We 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 variety of parameters including the system's uptime, the system's location, and information about TCP connections that are open on the system. Because we weren't able to get consistent code coverage measurements between runs of the same fuzzer and length, we ran each fuzzer and run length combination five times. The averages are displayed in our results, along with error bars for runs with noticeable variation (standard deviation of more than 0.025%).

Results. 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 rate-adjusted flow so that GPF would send the about the same number of packets LZFuzz sends for a given run length. With this adjustment we were able to have insight into how much influence a GPF-mutated packet has on the target compared to a LZFuzz-mutated packet. We also observed that LZFuzz induced a larger amount of code coverage in snmpd when the mutation rate that controls the mutation engine aggressiveness was set to *medium* (instead of high or low).[4] Because GPF uses the same mutation engine, we instructed GPF to run with a *medium* mutation level as well. Note that for snmpd, a 1% difference in code coverage corresponds to about 65 lines of code.

Figure 4 shows, on the right, the code coverage of GPF with a rate adjusted flow and a medium mutation rate, compared to LZFuzz with medium mutation, random fuzzing, and the environment with no fuzzer running. With rate adjusted flow, LZFuzz induces a higher code coverage than GPF. LZFuzz also clearly outperforms random fuzzing.

Comparing Apples to Oranges. Although LZFuzz and GPF share a common heuristic mutation engine, they fall in different classes of fuzzers, each with its own strengths and weaknesses. LZFuzz is capable of fuzzing both servers and clients; GPF can only fuzz targets that are listening to incoming traffic on a port known to GPF before the fuzzing session. GPF is capable of sending many packets in rapid succession; LZFuzz is restricted to only fuzzing packets sent by the source. GPF requires the user to spend time preparing a representative packet capture — and thus implicitly assumes such representative captures even exists for the target scenario. The time spent preparing the network capture is not taken into account in our results. The packet capture given to GPF potentially provides it with a wealth of information about the protocol from the get-go, whereas LZFuzz needs to develop most of its knowledge about the protocol it is fuzzing on the fly. Finally, the mutation engine of GPF was built and tuned specifically for what GPF does — fuzzing

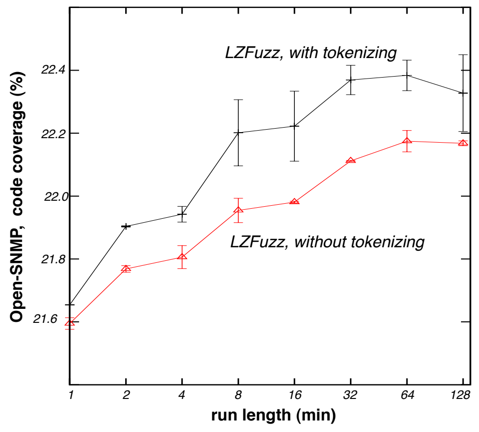

Figure 5. Comparison of code coverage of Net-SNMP server as produced by LZFuzz with tokenizing both enabled and disabled.

streams of packets. LZFuzz uses the same mutation engine but only ever has one packet in each stream. The GPF mutation engine is not designed to be used this way and we believe we can improve LZFuzz's effectiveness if we spent time tweaking the mutation engine.

Although when we used GPF and LZFuzz at full strength against mt-daap, LZFuzz outperformed GPF in terms of code coverage, we found this was not the case when both fuzzers were tested against snmmpd. When running at full-force, GPF was able to achieve 1-2% more code coverage than LZFuzz in comparable runs. We can argue that in the case of snmpd, GPF is the more effective fuzzer. However, the clear advantage of LZFuzz over GPF and similar fuzzers is that it is also capable of fuzzing SNMP clients, such as snmpwalk, whereas GPF is unable to do so without requiring some kind of session-tracking modifications.

4.4 Tokenizing

One might wonder whether the LZFuzz tokenizing method improves the overall effectiveness of LZFuzz. If tokenizing is disabled in LZFuzz during a run and the entire payload is passed directly to GPF, then GPF will attempt to perform its own heuristics to parse the packet. Figure 5 shows how LZFuzz with tokenizing compares to LZFuzz with tokenizing disabled when run against snmpd in the same environment as described in section 4.3. These results suggest that the LZ tokenizing does indeed improve the effectiveness of inline fuzzing with GPF's mutation engine.

5. Conclusion

In this paper we propose a novel approach to fuzzing that allows domain personnel without fuzzing expertise to efficiently fuzz implementations of proprietary protocols. We have shown that our adaptive live mutation fuzzing approach has the ability to fuzz the proprietary protocol, daap, more efficiently than other existing methods of fuzzing proprietary protocols. We have also shown the LZFuzz is more effective at fuzzing an SNMP server than random fuzzing. The GPF mutation fuzzer was more effective at fuzzing the SNMP server than LZFuzz; however unlike LZFuzz, GPF is unable to fuzz SNMP clients. Because LZFuzz requires minimal knowledge of the protocol it is fuzzing it can be used to empower SCADA asset owners to access the brittleness of their own equipment. LZFuzz has already been used to consistently crash a device in a SCADA test environment owned by a power partner.

Future Work. LZFuzz still needs more work to ensure it is a useful and usable appliance for a domain expert. Work needs to be done on the user interface so that the user can disable fuzzing and change the aggressiveness of the fuzzing on the fly, without restarting LZFuzz. Tools to make LZFuzz more useful include ones that attempt to identify checksums by intercepting traffic to the target and passively searching for bytes that appear to have a high entropy among the set of packets it sees. Another useful tool could test for the existence of authentication or connection setup traffic by inspecting traffic it sees at the beginning of a run as well as any traffic it sees from the target after blocking replies from the client, and vice versa. This information can be passed onto the domain expert so she can build custom traffic rules to make LZFuzz more effective.

Acknowledgments

We would like to acknowledge Axel Hansen and Anna Shubina's help on developing the initial prototype of LZFuzz [5]; we also would like to thank the unnamed power industry colleagues who let us test LZFuzz in an isolated test environment within their facilities.

This material is based upon work supported by the Department of Energy under Award Number DE-OE0000097. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

Notes

1. This test suite is not publicly available anymore.

2. There may be cases when there are hard crashes where the process does not automatically restart. In those cases the devices must be manually restarted.

3. LZFuzz may fuzz multiple tokens at a time in windowing mode in order to ensure there are enough bytes available to mutate effectively.

4. The mutation rate governs how much GPF's mutation engine mutates a packet. Although this feature isn't documented, this setting is required to be explicitly set during a fuzzing session. Line 143 of the GPF source file misc.c offers an option "MutationRate = Rate of Mutations: 'high', 'med', or 'low'" which it does not document further; we chose the 'med' option for this test and high' for mt-daapd.

References

[1] D. Aitel, An Introduction to SPIKE, the Fuzzer Creation Kit, *BlackHat USA*, 2002.
[2] P. Amini, PaiMei and the Five Finger Exploding Palm RE Techniques, *REcon 2006*, (`http://www.recon.cx/en/s/pamini.html`).
[3] arp-sk: A Swiss knife tool for ARP, (`http://sid.rstack.org/arp-sk/`).
[4] beSTORM Software Testing Framework, 2011 (`http://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, Dartmouth College Computer Science Technical Report TR2008-634, 2008 (`http://www.cs.dartmouth.edu/reports/TR2008-634.pdf`).
[6] Johnny Cache, H.D. Moore, skape, Exploiting 802.11 Wireless Driver Vulnerabilities on Windows, *Uninformed.org* vol. 6, 2007
[7] C. Cadar, V. Ganesh, P.M. Pawlowski, D.L. Dill, and D.R. Engler, EXE: Automatically Generating Inputs of Death, *ACM Transactions on Information and System Security,* 2008.
[8] Sean Convery, Hacking Layer 2: Fun with Ethernet Switches, *BlackHat USA*, 2002
[9] General Purpose Fuzzer, (`http://www.vdalabs.com/tools/efs_gpf.html`).
[10] ICCPSic, 2007 (`http://www.digitalbond.com/2007/08/28/iccpsic-assessment-tool-set-released/`).
[11] Dan Kaminsky, CFG9000 fuzzer,"Black Ops 2006", *BlackHat USA*, 2006
[12] H. Meer, Memory Corruption Attacks: The (Almost) Complete History, *BlackHat USA*, 2010
[13] B. Miller, L. Fredriksen, and B. So, An Empirical Study of the Reliability of UNIX Utilities, *Communications of the ACM*, 1990.
[14] C. Miller and Z. Peterson, *Analysis of Mutation and Generation-Based Fuzzing*, Independent Security Evaluators 2007 (`http://securityevaluators.com/files/papers/analysisfuzzing.pdf`).
[15] Mu Test Suite, 2011 (`http://mudynamics.com/products/Mu-Test-Suite.html`).
[16] C. Nevill-Manning and I. Witten, Sequitur, 1997, (`http://sequitur.info/`).
[17] Peach Fuzzing Platform, 2011 (`http://peachfuzzer.com/`).
[18] T. Pröll, Fuzzing Proprietary Protocols: A Practical Approached, *SecTor 2010.*
[19] QueMod, 2011 (`https://github.com/struct/QueMod`).
[20] J. Röning, M. Lasko, A. Takanen, and R. Kaksonen, PROTOS: Systematic Approach to Eliminate Software Vulnerabilities, Invited presentation at Microsoft, 2002
[21] Sulley Fuzzing Framework, 2011 (`http://code.google.com/p/sulley/`).
[22] J. Ziv and A. Lempel, A Universal Algorithm for Sequential Data Compression, *IEEE Transactions on Information Theory* 1977.