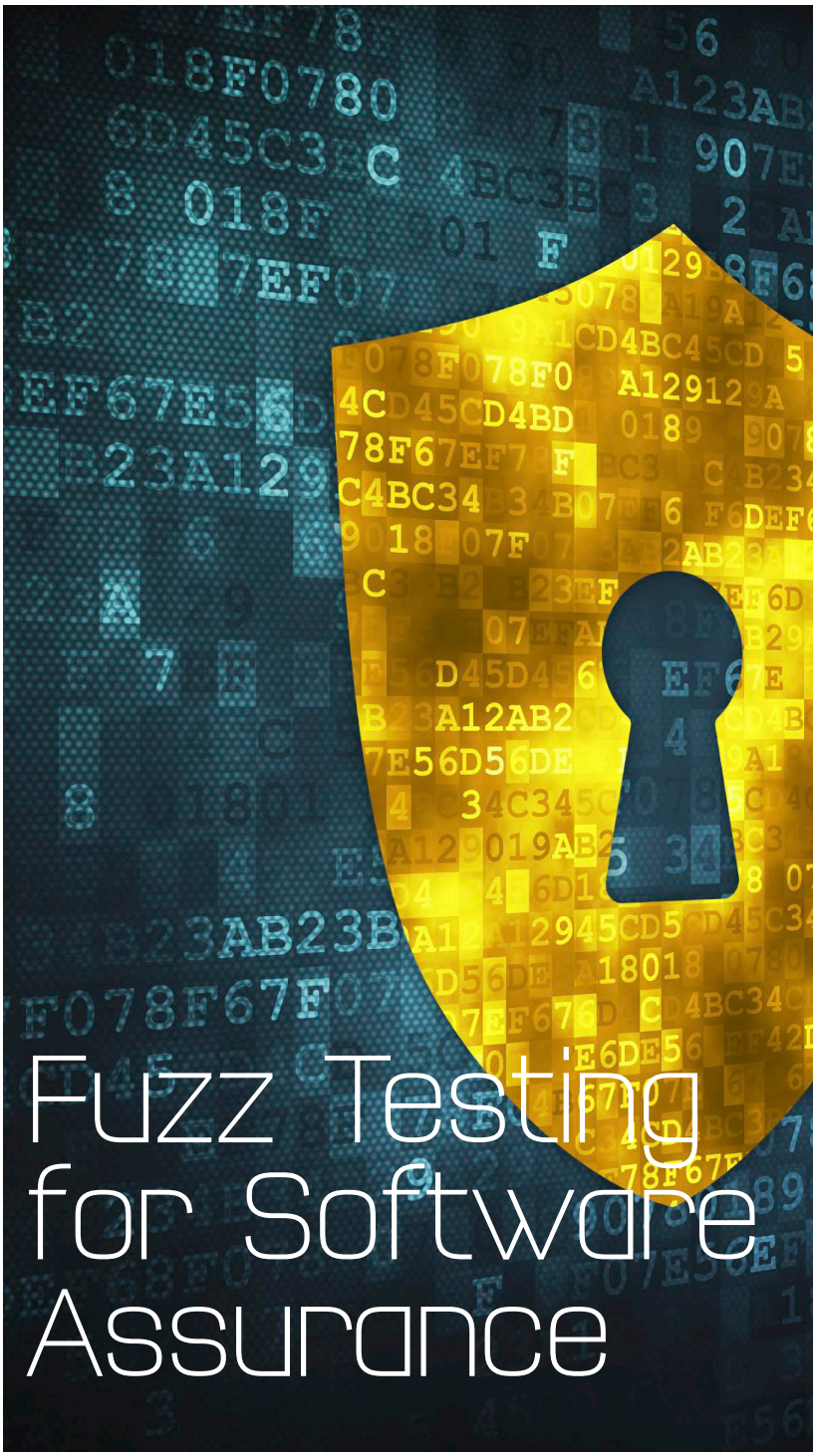

Fuzz Testing for Software Assurance

Vadim Okun, NIST
Elizabeth Fong, NIST

Abstract. Multiple techniques and tools, including static analysis and testing, should be used for software assurance. Fuzz testing is one such technique that can be effective for finding security vulnerabilities. In contrast with traditional testing, fuzz testing only monitors the program for crashes or other undesirable behavior. This makes it feasible to run a very large number of test cases. This article describes fuzz testing, its strengths and limitations, and an example of its application for detecting the Heartbleed bug.

Fuzz Testing and its Role for Software Assurance

Software assurance is level of confidence that software is free from vulnerabilities, either intentionally designed into the software or accidentally inserted at any time during its life cycle and that the software functions in the intended manner [1].

Multiple techniques and tools should be used for software assurance. Static analysis tools examine code for weaknesses without executing it. On the other hand, testing evaluates a program by executing it with test inputs and then compares the outputs with expected outputs. Both static analysis and testing have a place in the software development life cycle.

Positive testing checks whether a program behaves as expected when provided with valid input. On the other hand, negative testing checks program behavior by providing invalid data as input. Due to time constraints, negative testing is often excluded from the software development life cycle. This may allow vulnerabilities to persist long after release and be exploited by hackers. Fuzz testing is a type of negative testing that is conceptually simple and does not have a big learning curve.

Fuzz testing, or fuzzing, is a software testing technique that involves providing invalid, unexpected, or random test inputs to the software system under test. The system is then monitored for crashes and other undesirable behavior [2].

The first fuzzing tool simply provided random inputs to about 90 UNIX utility programs [3]. Surprisingly, this simple approach led to crashes or hangs (never-ending execution) for a substantial proportion of the programs (25 to 33%).

Fuzz testing has been used to find many vulnerabilities in popular real-life software. For example, a significant proportion of recent vulnerabilities in Wireshark (http://www.wireshark.org), a network protocol analyzer, were found by fuzzing. Large organizations are taking note. For example, Microsoft includes fuzz testing as part of its Security Development Lifecycle (http://www.microsoft.com/security/sdl/default.aspx).

A fuzzing tool, or fuzzer, consists of several components and a fuzzing process involves several steps [4]. First, a generator produces test inputs. Second, the test inputs are delivered to the system under test. The delivery mechanism depends on the type of input that the system processes. For example, a delivery mechanism for a command-line application is different from one for a web application. Third, the system under test is monitored for crashes and other basic undesirable behavior.

Strengths and Limitations of Fuzz Testing

Fuzz testing is conceptually simple and may offer a high benefit-to-cost ratio. In traditional testing, each test case consists of an input and the expected output, perhaps supplied by an oracle. The output of the program is compared to the expected output to see whether the test is passed or failed. In the absence of executable specifications or a test oracle (e.g. a reference implementation or checking procedure), finding the expected output for a lot of test cases can be costly. In contrast, fuzz testing only monitors the program for crashes or other undesirable behavior. This makes it feasible to run hundreds of thousands or millions of test cases.

Fuzz testing is effective for finding vulnerabilities because most modern programs have extremely large input spaces, while test coverage of that space is comparatively small [5].

While static source code analysis or manual review are not applicable to systems where source code is not available, fuzz testing may be used. Fuzz testing is a general technique and therefore may be included in other testing tools and techniques such as web application scanners [6].

Fuzz testing has a number of limitations [7]. First, exhaustive testing is infeasible for a reasonably large program. Therefore, typical software testing, including fuzz testing, cannot be used to provide a complete picture of the overall security, quality or effectiveness of a program in any environment. Second, it is hard to exercise the program thoroughly without detailed understanding, so fuzz testing may often be limited to finding shallow weaknesses with few preconditions. Third, finding out what weakness in code caused the crash may be a time-consuming process. Finally, fuzz testing is harder to apply to categories of weaknesses, such as buffer over-reads, that do not cause program crashes.

Fuzzing Approaches

Test input generation can be as simple as creating a sequence of random data [3]. This approach does not work well for programs that expect structured input. No matter how many tests are generated, the vast majority might only exercise a small validation routine that checks for valid input.

In regression testing, valid inputs may be collected, for example, from historical databases of unusual inputs that caused errors in the past versions of the software, and then supplied to the program without modification. Such approach can help uncover a weakness that reoccurs between versions or implementations, but is unlikely to uncover new weaknesses.

Most fuzz generators can be divided into two major categories: mutation based and generation based fuzzers [8]. A mutation based fuzzer produces test inputs by making random changes to valid test input, such as those from regression testing. This approach can be quickly applied to systems, such as protocols or word processors, that accept complex inputs. However, the coverage is only as strong as the set of valid test inputs. If there is no valid test input for a particular system component, the mutation based fuzzer is unlikely to cover this component.

A generation based fuzzer produces test inputs based on some specification of the input format. While implementing the input format in enough detail requires a significant upfront effort, the generation based fuzzer can achieve very high coverage at lower cost.

A relatively recent approach, whitebox fuzz testing, combines symbolic execution with constraint solving to construct new inputs to a program [9]. Whitebox fuzzing has been used by Microsoft to find one third of all the bugs discovered by file fuzzing during the development of Windows 7.

The next step after producing test inputs is providing them to the system under test. Some common delivery mechanisms are files, environment variables, command line and API parameters, and operating system events, such as mouse and keyboard events.

Fuzz testing does not require knowing the expected output, instead there must be monitoring for crashes or other generally undesirable behavior. However, many types of weaknesses do not produce clearly undesirable behavior. Therefore, more sophisticated detection that test input caused a failure can significantly expand the classes of weaknesses uncovered by fuzzing. The following Section describes an example of using dynamic analysis tools to detect a weakness that does not cause a crash under normal operation.

Protocol Testing Experiment

The Heartbleed bug is a widely known vulnerability in OpenSSL, a popular implementation of the cryptographic protocols Secure Sockets Layer (SSL) and Transport Layer Security (TLS). Briefly, under the Heartbeat protocol, the client sends a message and the message length to the server, and the server echoes back the message.

The Heartbleed vulnerability can be exploited to leak confidential information, including passwords and authentication data. It was caused by the failure of OpenSSL to validate the message length, which caused Buffer over-read weakness [10]. For more details, an interested reader can examine Heartbit, an abstracted version of the OpenSSL code demonstrating the Heartbleed vulnerability [11]. Even though buffer overflow, which includes buffer over-read, is a well-known weakness, software assurance tools missed it [12].

Simple fuzz testing, which looks for crashes, would not have detected Heartbleed. The reason is that buffer over-reads rarely lead to program crashes. However, fuzz testing in combination with a memory error detection tool, may have detected Heartbleed, as demonstrated in [13].

Memory error detection tools, such as Valgrind (http://valgrind.org) and AddressSanitizer (http://code.google.com/p/address-sanitizer), are a type of dynamic analysis tools that can be used to instrument code to detect various memory errors, such as buffer overflows and use-after-free errors that may not cause a crash under normal operation.

In the first experiment, [13] ran a vulnerable version of OpenSSL with Valgrind. When the fuzzer sent an exploiting Heartbleed request, Valgrind produced an error trace highlighting the bug. In the second experiment, a vulnerable version of OpenSSL was compiled with the AddressSanitizer compiler option. When an exploiting Heartbleed request was sent to the server, it terminated and an error trace was produced. In both experiments, a programmer could use the error trace to find the Heartbleed bug.

Conclusions

Typical software testing, including fuzz testing, cannot be used alone to produce bug-free software. Since fuzz testing does not require a sophisticated oracle, it can quickly test a very large number of unexpected inputs. When combined with appropriate supplemental tools, this makes it possible to find security vulnerabilities, such as the Heartbleed bug, which may be missed by other tools. As demonstrated by a large number of bugs recently

discovered in production software, fuzz testing can be used to increase software assurance.

Disclaimer:

Any commercial product mentioned is for information only; it does not imply recommendation or endorsement by NIST nor does it imply that the products mentioned are necessarily the best available for the purpose.

ABOUT THE AUTHORS

Vadim Okun is a Computer Scientist at the National Institute of Standards and Technology, where he is leading the SAMATE (http://samate.nist.gov/) team. His current research focuses on software assurance, in particular, the effect of tools on security. He organized Static Analysis Tool Expositions (SATE) – large-scale evaluations to support research in, and improvement of, static analysis tools. Previously, Okun contributed to the development of automated software testing methods: specification-based mutation analysis and combinatorial testing. He received a Ph.D. in Computer Science from University of Maryland Baltimore County.

E-mail: vadim.okun@nist.gov

Elizabeth Fong is a computer scientist currently working in the Software and Systems Division, Information Technology Laboratory of National Institute of Standards and Technology. She performs research, analysis, and evaluation of innovative information technology software and practices with emphasis on new fields of computer technology for Federal Government applications. Recent work involved software assurance, smart card technology, XML registry framework and standards, and interoperability testing technology. Earlier work included technologies and standards development for Object-Oriented, distributed database management and agent-based computing. She has many years of experience in the development of software testing and reference models for information technologies. She received B.SC (Math) New York University, New York, NY, M.S. (Computer Science) Stanford University, CA and Graduate Courses (Computer Science) U. of Maryland, MD.

E-mail: efong@nist.gov

REFERENCES

1. CNSS, National Information Assurance (IA) Glossary CNSSI-4009, 26 April 2010, p. 69, http://www.ncix.gov/publications/policy/docs/CNSSI_4009.pdf.
2. Wheeler, David A. and Rama S. Moorthy, "SOAR for Software Vulnerability Detection, test and Evaluation," IDA paper P-5061, July 2014.
3. Miller, Barton P., Lars Fredriksen, and Bryan So, "An Empirical Study of the Reliability of UNIX Utilities," Communications of ACM 33(12):32-44 (Dec. 1990).
4. McNally, R., Yiu, K., Grove, D., and Gerhardy, D., "Fuzzing: The State of the Art," Technical Note DSTO-TN-1043, 2012.
5. Householder, Allen D., "Why Fuzzing (Still) Works," in Metrics and Standards for Software Testing (MaSST) workshop, p. 39-59, December 2012, http://samate.nist.gov/docs/MaSST_2012_NIST_IR_7920.pdf.
6. Fong, Elizabeth, Romain Gaucher, Vadim Okun, Paul E. Black, and Eric Dalci, "Building a Test Suite for Web Application Scanners," 41st Hawaii Int'l Conf. on System Sciences (HICSS), January 2008.
7. West, Jacob, "How I Learned to Stop Fuzzing and Find More Bugs," DefCon, Las Vegas, August 2007.
8. Neystadt, John, "Automated Penetration Testing with White-Box Fuzzing," http://msdn.microsoft.com/en-us/library/cc162782.aspx, Microsoft, February 2008.
9. Bounimova, Ella, Patrice Godefroid, and David Molnar, "Billions and Billions of Constraints: Whitebox Fuzz Testing in Production," ICSE 2013:122-131.
10. Wheeler, David A., "How to Prevent the next Heartbleed," http://www.dwheeler.com/essays/heartbleed.html, 2014.
11. Bolo – Joseph T. Burger, "Heartbit test case," http://samate.nist.gov/SARD/view_testcase.php?tID=149042.
12. Kupsch, James A. and Miller, Barton P. "Why Do Software Assurance Tools Have Problems Finding Bugs Like Heartbleed?" Continuous Software Assurance Marketplace, 22 Apr. 2014, https://continuousassurance.org/swamp/SWAMP-Heartbleed.pdf.
13. Vassilev, Apostol and Christopher Celi, "Avoiding Cyberspace Catastrophes through Smarter Testing," IEEE Computer, October 2014; 47(10):86-90.