Fuzzing for Software Security Testing and Quality Assurance

Ari Takanen
Codenomicon Ltd
ari.takanen@codenomicon.com

Abstract

Fuzzing is a proactive method for discovering zero-day security flaws in software. Fuzzing can be used in R&D, but also when deploying communication software. The system under test can be an enterprise solution, or it can be a consumer product such as a mobile phone or a set-top box for IPTV. Without proactive tools, the traditional security measures are doomed to fail because they are only focused on defending from known attacks. Based on our experience with both academic and commercial fuzzing tools, we can show that 80% of software will crash when tested with negative testing. We will explain the history of fuzzing and the different technologies used by different fuzzers. The tools are broken into logical functions, and each function is explained. Attention is also paid on various metrics related to fuzzing, to enable comparison of efficiency of various tools.

Reactive versus Proactive

Security has traditionally been reactive, focused on defending from attacks. A proactive approach should focus on fixing the actual flaws enabling these attacks. An attack does not work if there is no vulnerability. Majority of flaws reported publicly are found by third parties, and require expensive and time-sensitive process for disclosure of the vulnerability data, building of corrections (patches) and distribution/deployment of the corrective measures. A bug found as part of the software development process will not go through this extensive process but is handled just like any other critical flaw in the system.

Fuzzing as a Proactive Measure

Proactive security testing approaches include fuzzing, protocol mutation, robustness testing, and the like. Especially fuzzing is a very effective way of discovering software vulnerabilities, as it requires no intelligence of the internal operations of the device or system under test. Legacy fuzzing was based on randomly mutated inputs (white-noise testing), and was only used by professional security specialists and selected researchers. But today most fuzzers are based on intelligent model-based test automation techniques. Besides making the use of fuzzing tools much easier, this enables much higher vulnerability detection rates.

Automation equals Efficiency

Fuzzing is a rather new test automation technique for finding critical security problems in any type of communication software. It is a negative software testing method (negative testing) that feeds a program, device or system with malformed and unexpected input data in order to find critical crash-level defects. The tests are targeted at remote interfaces, but can also test local interfaces and API. Focus on most critical remote interfaces typically means that fuzzing is able to cover the most exposed and critical attack surfaces in a system relatively well, and identify many common errors and potential vulnerabilities quickly and cost-effectively. Only recently, it was mostly an unknown hacking technique that very few quality assurance specialists knew about. Today, both quality assurance engineers and security auditors use fuzzing. It is a mainstream testing technique used by all major companies building software and devices for critical communication infrastructure.

Focus on Finding Vulnerabilities

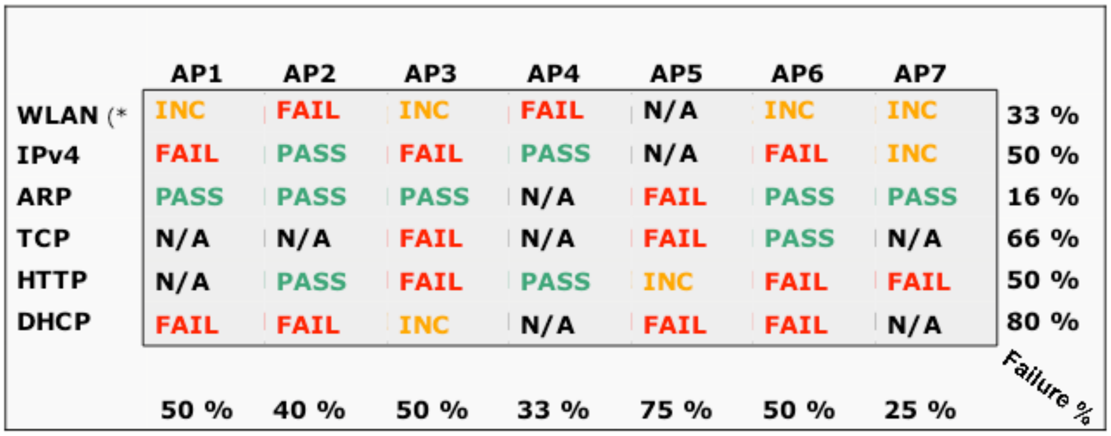
Fuzzing is focused on detecting implementation issues in software. Vulnerability databases indicate that programming errors causes 80% of the publicly known vulnerabilities. Inclusion of the vulnerabilities caught in the software development would probably increase this even further. Today, no more than 25% of vulnerabilities are found with the traditional software quality assurance processes, with majority of the software companies catching less than 5% of the vulnerabilities in hiding. Static analysis tools cannot be used in post-release testing, or in third party security analysis. Improving the software testing practices can eliminate these worm-size holes before product launch, and without requiring access to the source code.

Evolution of Fuzzing

The term ‘fuzzing’ or ‘fuzz testing’ emerged around 1990, but in its original meaning fuzzing was just another name for random testing, with very little use in Quality Assurance (QA) beyond some limited ad-hoc testing. Still, the transition to integrating the approach into software development was evident even back then. During 1998-2001, in the PROTOS project (at University of Oulu) we conducted research that had a focus on new model-based test automation techniques, and other next generation fuzzing techniques [2]. The purpose was to enable the software industry themselves to find security critical problems in a wide range of communication products, and not to just depend on vulnerability disclosures from third parties. Codenomicon is a spin-off from the project, founded in 2001, that today continues to lead the state-of-the-art in fuzzing techniques. Later, around year 2007 also other companies became fascinated in the topic, and a new highly competed test and measurement market domain was created. Finally in 2008, among several other books on the topic, we also released a book (with the help of other fuzzing specialists) which gives a broader and but also more detailed look on how fuzzing can be used in different steps in the software lifecycle [5]. This started the move towards making fuzzing a best practise in software development.

Attack Surface

Communications function in layers of protocols. Data is transmitted in protocols running on top of each other, and much of the security analysis only looks at the highest layers, ignoring protocol level attacks. Fuzzing can be conducted on all layers of communications. Security research focusing on one layer fuzzing only (such as the publicly available PROTOS research results) indicate that more than 80% of all software will crash when tested with negative testing. Additional studies where more layers were included show that any interface on any communication device taken for analysis has less than 10-30% survival rate, with majority of devices failing in more than half of the interface tests. With a sample of products under tests, no device was found to be secure on all layers of communication (Figure 1).



	AP1	AP2	AP3	AP4	AP5	AP6	AP7	
WLAN (*	INC	FAIL	INC	FAIL	N/A	INC	INC	33 %
IPv4	FAIL	PASS	FAIL	PASS	N/A	FAIL	INC	50 %
ARP	PASS	PASS	PASS	N/A	FAIL	PASS	PASS	16 %
TCP	N/A	N/A	FAIL	N/A	FAIL	PASS	N/A	66 %
HTTP	N/A	PASS	FAIL	PASS	INC	FAIL	FAIL	50 %
DHCP	FAIL	FAIL	INC	N/A	FAIL	FAIL	N/A	80 %
	50 %	40 %	50 %	33 %	75 %	50 %	25 %	Failure %

Figure 1: Test results from fuzzing seven WiFi access points against six different interfaces/protocols

Test Efficiency

Comparing fuzzing tools is difficult, and there is no accepted methodology for that. The easiest method is based on the enumeration of interface requirements. One toolkit might support about 20 or so protocol interfaces where another one will cover more than 100 protocols. Testing a web application requires a different set of fuzzers than testing a voice over IP (VoIP) application. Some fuzzing frameworks are powerful at testing simple text-based protocols, but provide no help in testing complex structures such as ASN.1 or XML. Other fuzz-tests come in pre-packaged suites around common protocols such as SSL/TLS, HTTP, and UPnP, whereas in other cases you need to build the tests yourself. The test direction and physical interfaces can also impact the usability of some tools, as some of them only test server side implementations in a Client-Server infrastructure.

In one research for studying fuzzing test efficiency [5], fuzzers were compared by running their tests against a piece of software that had intentionally planted security vulnerabilities in it. Selected protocol interfaces and open source implementations were selected, and common vulnerabilities were created to the implementations. The fuzzer efficiency was noted to range from 0% up to 80%. The detection rates of fuzzers ranged from 10-30% for random fuzzers and up to 70-80% for model-based fuzzing tools. Majority of the security problems intentionally created for the open source implementations were caught and some previously unknown flaws were also detected, adding up to an estimate of 90% detection rate through fuzzing. The tool with most test cases very rarely was the most efficient one, so looking at the number of test cases will often lead into selecting a tool that has least intelligence in the test generation. The testing technique had no false positives, as each found issue was truly a remotely exploitable weakness in the tested software.

Product Security

Recent studies [3] indicate that all major software security initiatives have included fuzzing in their activities. Market studies by leading analysts and reports by leading commercial fuzzing companies indicate that the adoption of fuzzing is still in rapid growth. Users of fuzzing tools come from all industries, and from all parts of the organizations. Some users look at fuzzing as a quality assurance technique. Others use fuzzing in penetration testing. And finally leading Enterprises have integrated fuzzing in acceptance testing and procurement processes. The range of use cases for fuzzing is growing fast, and integration capabilities of fuzzing tools are improving as the use cases become clearer.

Software Requirements

To understand the principles behind fuzzing, we need to look how it fits into the entire software lifecycle. As the software development process starts from requirements gathering, so let's first look at how the requirements for security and fuzzing can be mapped together. A software requirements specification often consists of two different types of requirements (Figure 2). Firstly, you have a set of positive requirements that define how software should function. Secondly, you have a set of negative requirements that define what software should not do. The actual resulting software is a cross-section of both of these. Acquired features, and conformance flaws, map against the positive requirements. Fatal features and unwanted features map into the negative requirements. The undefined grey area in-between the positive and negative requirement leaves room for the innovative features that never made it to the requirements specifications or to the design specifications, but were implemented as later spontaneous decisions (or by mistake) during the development. These are often difficult to test, as they might not make it to the test plans at all. The main focus of fuzzing is not to validate any correct behaviour of the software but to explore the challenging area of negative requirements. [1]

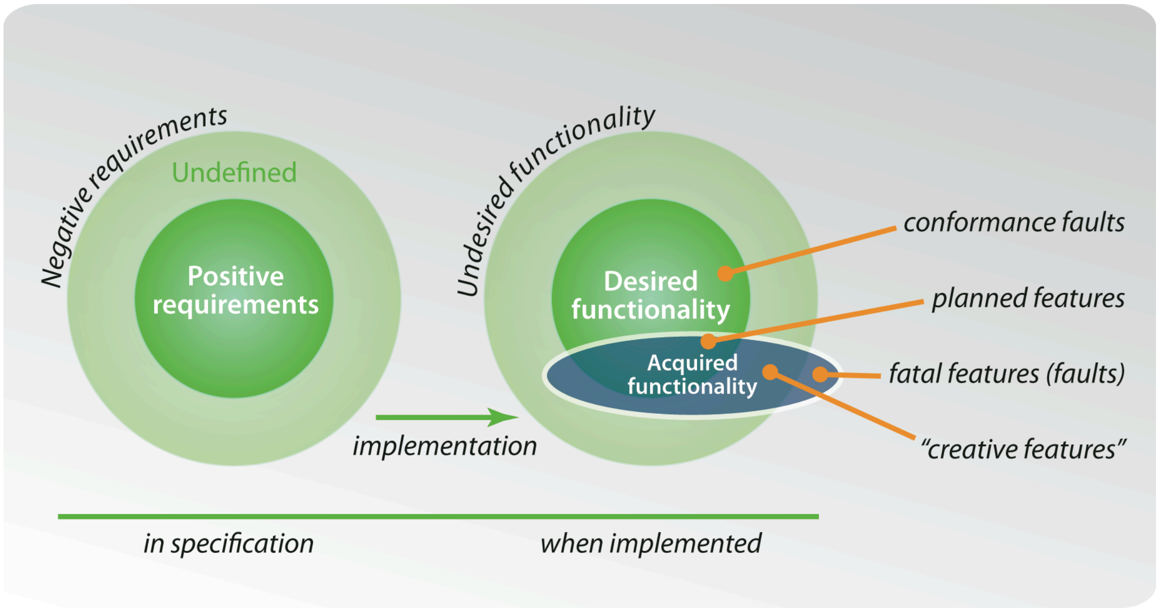


Figure 2: Mapping requirements to flaws in delivered software [5]

Types of Testing

Looking at different types of black-box testing, we can identify three main categories of testing techniques (Figure 3). These are feature testing, performance testing, and robustness testing. Feature testing is the traditional approach of validating and verifying functionality, whereas performance testing looks at the efficiency of the built system. Both feature testing and performance testing are exercising the system with valid inputs. Robustness testing on the other hand tests the system under invalid inputs, focusing on checking the system stability, security and reliability. Comparing these three testing categories, we can note that most feature tests map one-to-one against use-cases in the software specifications. Performance testing on the other hand uses just one use-case, but loops that either in a fast loop, or in multiple parallel executions. In robustness testing, you build thousands or sometimes even millions of misuse-cases for each use-case. Fuzzing is one form of robustness testing, focusing on the communication interfaces and discovery of security related problems such as overflows and boundary value conditions, in order to more intelligently test the infinite input space that is required to try out in robustness testing.

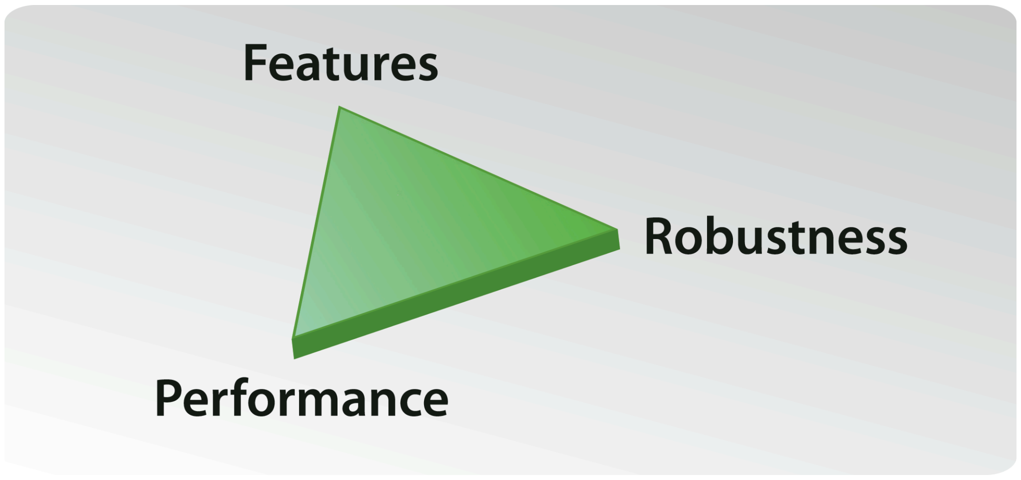


Figure 3: Three types of blackbox testing: testing for features, performance and robustness [5]

Cost of Bugs

The purpose of fuzzing is to find security critical flaws, and the timing of such test will have heavy impact on the total cost of the software. Therefore the most common view in analyzing fuzzing benefits is looking at costs related to finding and fixing security related bugs. Software product security has a special additional attribute to it, as most of the costs are actually caused to the end user of the software from the software maintenance, patch deployment, and damages from incidents. The security compromises, or denial of service incidents impact the users of the software, not the developers of software. This is why the cost metrics often include both the repair costs for the developers, and the costs from damages to end-users. These are often the very same metrics that you might have developed for analyzing the needs for static analysis tools. Depending on which phase of the software lifecycle you focus your testing efforts, the cost per bug will change. If you can find and fix a problem early in the product lifecycle, the cost per bug is much less compared to a flaw found after the release of the software. This type of analysis is not easy for static analysis tools due to the rate of false positives, indicated flaws that do not have any significance from security perspective. A metric collected early in the process might not give any indication of the real cost saving. This is different for fuzzing. Whereas static analysis tools create poor success rate based on analyzing the real security impact of the found flaws, there are no false positives in fuzzing. All found issues are very real, and will provide a solid metric for product security improvements.

Uses of Fuzzing

Although originally fuzzing was mainly a tool for penetration testers and security auditors, today the usage is much more diverse. Soon after the exposure caused by PROTOS, most network equipment manufacturers (NEMs) quickly adapted the tools into their quality assurance processes, and from that the fuzzing technologies evolved into quality metrics for monitoring the product lifecycle and product maturity. Perhaps because of the rapid quality improvements in network products, fuzzing soon also became a recommended purchase criterion for enterprises, pushed by vendors who were already conducting fuzzing and thought that it would give them a competitive edge. As a result, service providers and large enterprises started to require fuzzing and similar testing techniques from all their vendors, further increasing the usage of fuzzing. In short, today fuzzing is used in three phases at the software lifecycle:

1. QA Usage of Fuzzing in Software Development
2. Regression testing and product comparisons using Fuzzing at test laboratories
3. Penetration testing use in IT operations

As the usage scenarios range from one end to another, so does the profile of the actual users of the tools. Different people look for different aspects in fuzzers. Some users prefer random fuzzers whereas others look for intelligent fuzzing. Other environments require appliance-based testing solutions, and other test environments dictate software-based generators. Fortunately all those are easily available today.

Test Automation

Let's then review how fuzzing maps to different test automation techniques. Different levels of test automation are used in all testing taking place today, and fuzzing is just one additional improvement in that domain (Figure 4) [4].

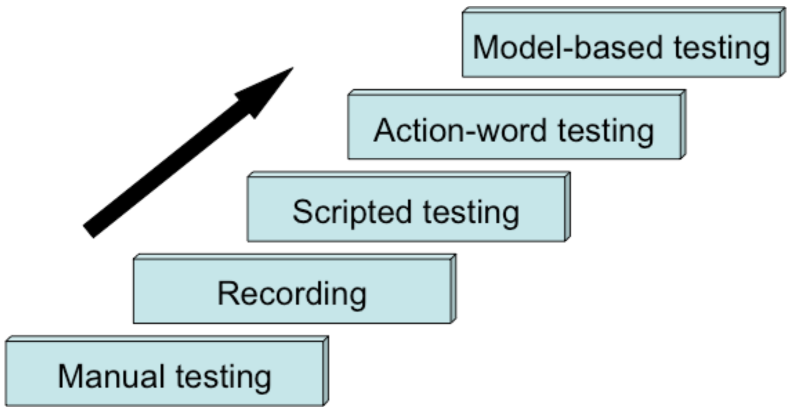


Figure 4: Test automation maturity levels (based on presentation by Olli-Pekka Puolitaival [4])

In fact, test automation experts are often the first people that familiarize themselves with fuzzing and other related test generation techniques. Test automation often focuses only on the repeatability of tests, but another significant improvement from some test automation techniques is the improvements in test design and test efficiency. The more advanced tools you use, the less work is required for the entire testing cycle. The most basic transition into test automation is moving from manual testing into test recording and replaying methodologies. The most extreme developments in test automation are in model-based testing. In fact, the laborious test design phase is mostly skipped in model-based testing, as the tests are automatically generated from system models and communication interface specifications.

Test Automation in Fuzzing

Not all fuzzing tools are model-based, but all types of fuzzing techniques are always highly automated. When people conduct fuzzing, there is close to zero human involvement in the testing. In fuzzing, the tests are automatically generated, executed, and the test reporting is also automated so that most of the work can be focused on analyzing and fixing the found issues.

Capture-Replay versus Full Model

Two different test automation techniques are popular in fuzzing. The major difference is in where the "model" of the interface is acquired. The easiest method of building a fuzzer is based on re-using a test case from feature testing or performance testing, whether it is a test script or a captured message sequence, and then augmenting that piece of data with mutations, or anomalies. The simplest form of mutation fuzzing is based on just randomly doing data modifications such as bit flipping and data insertion, in order to try unexpected inputs. The other method of fuzzing is based on building the model from communication protocol specifications and state-diagrams.

Mutation-based fuzzers break down the structures used in the message exchanges, and tag those building blocks with meta-data that helps the mutation process. Similarly, in full model-based fuzzers each data element needs to be identified, but that process can be also automated as that information is often already given in the specifications that are used to generate the models. Besides information on the data structures, the added meta-data can also include details such as the boundary limits for the data elements.

In model-based fuzzing the test generation is often systematic, and involves no randomness at all. Although many mutation and block-based fuzzers often claim to be model-based, a true model-based fuzzer is based on a dynamic model that is "executed" either runtime or off-line. In PROTOS research papers, this approach of running a model during the test generation or test execution was called Mini-Simulation [2]. The resulting executable model is basically a full implementation of one of the end-points in the communication.

Conclusion

Fuzzing is here to stay, and if you will not use it yourself, someone else will. Fuzzing tools are easily available as well supported commercial solutions but also as free open source solutions. Not doing fuzzing at all could be considered negligence. Fuzzing is a very efficient method of finding remotely exploitable holes in critical systems, and the return of time and effort placed in negative testing is immediate. Just one flaw when found after the release can create enormous costs through internal crisis management, and compromises of deployed systems. No bug will stay hidden if correct tools are used. Still, there is room for development in fuzzing research, and we are happy to see that research teams are embracing fuzzing as a new security research topic.

In this paper (and related panel talk) we focused in use of fuzzing in software development. As part of that we looked at the brief history of fuzzing and the different technologies used by different fuzzers. The techniques behind different types of fuzzers were explained to give the reader an understanding on how different fuzzers work. By studying the use cases of fuzzing we aimed to assist the audience in integrating fuzzing in their own software development lifecycle. Attention was also paid on various metrics related to fuzzing, to enable comparison of efficiency of various tools. We also explained various aspects of test automation to help you map this proactive security testing technique to other quality assurance practises.

References

[1] Eronen, Juhani; Laakso, Marko: A Case for Protocol Dependency. In proceedings of the First IEEE International Workshop on Critical Infrastructure Protection. Darmstadt, Germany. November 3-4, 2005.

[2] Kaksonen, Rauli: A Functional Method for Assessing Protocol Implementation Security (Licentiate thesis). Espoo. 2001. Technical Research Centre of Finland, VTT Publications 447. 128 p. + app. 15 p. ISBN 951-38-5873-1 (soft back ed.) ISBN 951-38-5874-X (on-line ed.).

[3] McGraw, Gary; Chess, Brian and Migues, Sammy: Building Security In Maturity Model. www.bsi-mm.com, 2009, 53 p.

[4] Puolitaival, Olli-Pekka: Model-based testing tools. Presentation at Software Testing Day at TUT. March 25-26, 2008.

[5] Takanen, Ari; DeMott, Jared and Miller, Charlie: Fuzzing for Software Security Testing and Quality Assurance. Artech House. 2008. 230p. ISBN 978-1-59693-214-2.