



MIKKO VARPIOLA,
ARI TAKANEN

How to Deploy Robustness Testing

Difficulty



Today's software companies design and test their code using the well-accepted, familiar method of positive testing. Still, all communications software appears to be infested with security-critical bugs that can be misused to crash the software or to take total control of the device running the software.

More than 80 percent of all attack tools exploit these types of simple implementation mistakes. And everything fails when tested for these types of flaws. In this article, we will explore various means of testing for such security mistakes, with a focus on deploying robustness testing into the software development lifecycle.

Positive Testing for Conformance

Positive testing encompasses all testing approaches that aim to validate features against requirements documentation. A simple requirement could say, for example, *Software must prompt for username and password before granting access to the data*. At a later point in the software development lifecycle, a test designer will read the requirement and implement a number of test cases to validate the feature. In the example, the designer would likely try, at minimum, to input the correct username and password.

While positive testing is prevalent in the software design cycle, it does have drawbacks. Programmers usually receive a rough guideline of vague, incomplete specifications. The same incomplete requirements make it difficult for test designers to consider the infinite number of permutations of test cases required to validate each feature. As most test design in positive testing is conducted through manual work, reaching any good test case coverage will take an enormous amount of time, delaying the product launch. Furthermore, the claim of 100

percent test coverage does not typically account for the complex tests required to catch unexpected error situations. That is where robustness or negative testing comes into play.

Negative Testing Through Randomness or Protocol Models

Negative testing, or fuzzing, extrapolates the complex interactions far beyond regular feature testing. The main objective of the negative testing methods is to identify the most critical problems so that programmers can improve software quality, reducing public exposure to crash-level software defects. The challenge in negative testing is that for each positive requirement, there is an infinite number of negative test cases that need to be tried (Figure 1). Random testing is one method of validation, but it has very little chance of discovering security-related flaws. Negative testing can accomplish this goal effectively through systematic grammar-based test generation.

A random test for the example given earlier would aim to generate a predefined number of tests or an undefined number of tests in a predefined time, using pairs of random usernames and passwords. The entire test run for the validation would probably consist of two test groups, each containing a huge number of individual test cases. For any effective testing, at least part of the *protocol* needs to be valid, and therefore the first test suite would keep the username valid and randomly corrupt the

WHAT YOU WILL LEARN...

How robustness/negative testing fills the critical gap left by positive testing – and reduces the cost and time required for testing

Results of comparing positive vs. negative testing and random fuzz testing vs. model-based negative testing

How to analyze test coverage in a way that specifies which flaws were exposed

WHAT YOU SHOULD KNOW...

Basic principles of fuzz testing, AQ and the difference between model-based testing versus random fuzz testing

Fundamental concepts of negative testing versus positive testing

password, and then in the next test suite the username would be corrupted.

In model-based or grammar-based negative testing, both username and password are enumerated through a predefined set of anomalies in an anomaly library. Based on past vulnerability knowledge, the anomaly library is highly optimized to catch different types of problems in code. Example tests could include commonly used passwords, long strings, format string problems, boundary value conditions and so on. A model-based suite of negative tests can consist of tens of thousands of carefully thought-out test cases. From a test documentation perspective, it is typically considered one test suite, or a test, and the pass/fail criteria for the test are given for each test run. For example, a bad test document can say that all external interfaces need to be tested for security problems whereas a good test documentation will define the interfaces and interface specifications that need to be covered in the test. Still, each failed test run against a specific interface and its specification can uncover a large number of faults mapping down to a set of flaws in the product (Figure 2). Both negative testing approaches have their pros and cons. A random fuzz test is extremely easy to implement and run. The simplest method for random fuzzing is to send random garbage to each open network interface. A slightly better method of conducting random testing is to take a template use case, a positive test case, and start semi-randomly mutating it. Random testing is the minimum amount of negative testing that designers should do for every open interface and network protocol. It will usually crash the software – The shortcoming is that the test coverage is almost impossible to estimate.

Metrics for Fuzzing

Test coverage is the first question a quality assurance expert will want to know when negative testing is described to him. What was tested and what was untested? Did we find all flaws, or just five percent of the flaws? This can be challenging for random testing, but also for template based fuzz tests. Model-based fuzzing will answer these questions a bit better, although it still cannot guarantee that all flaws were found. The quality of model-based fuzzing is based on the quality of the model, and the vulnerability

knowledge integrated into the used tool. The challenge in model-based testing is that building a good model often takes a lot of protocol and vulnerability knowledge.

One method of measuring the test efficiency of negative testing is to use code coverage tools. This can be done with open source tools such as *gcov*, which requires access to the source code for instrumenting the code with coverage hooks. Runtime coverage tools such as *PaiMei* will analyze the binary while it is being executed and report the coverage based on the instruction pointer as it walks through the executable code.

Comparisons Through Code Coverage

In our studies related to code coverage between positive tests and negative tests,

we saw that negative testing can reach code that will not be touched by positive tests alone. An example of such code would be error handling routines and other exception handlers. Table 1 shows results of running a mode-based fuzzer and a conformance test suite against OpenSSL implementation. Although these results are from a Master's Thesis study by Tero Rontti in 2004. Even back then, they did a revealing discovery on how negative testing will explore more code compared to positive testing alone. The coverage was mostly overlapping, although some of the code was touched only by one or the other testing technique (Table 1).

A comparison of random testing versus model-based fuzzing is even more revealing. A fuzzer product that

test-group	in...	status	inp...	inp...	out...	out...	inst...	diagn...	time
http11-accept...	4276	n/a	0	0	1	350	1	passed	0.371
http11-accept...	4277	n/a	0	0	1	350	1	passed	0.27
http11-accept...	4278	n/a	0	0	1	350	1	passed	0.361
http11-accept...	4279	n/a	0	0	1	350	1	passed	0.19
http11-accept...	4280	n/a	0	0	1	350	1	passed	0.31
http11-accept...	4281	n/a	0	0	1	350	1	passed	0.291
http11-accept...	4282	n/a	0	0	1	350	1	passed	0.26
http11-accept...	4283	n/a	0	0	1	350	1	passed	0.231
http11-accept...	4284	n/a	0	0	1	350	56	denial...	0.327
http11-accept...	4285	n/a	0	0	1	350	1	passed	0.34
http11-accept...	4286	n/a	0	0	1	350	1	passed	0.231
http11-accept...	4287	n/a	0	0	1	350	1	passed	0.31
http11-accept...	4288	n/a	0	0	1	350	1	passed	0.321

Figure 2. The results of a test run with one single input causing a failure (crash) among hundreds of other tests (Source: Codenomicon HTTP Test Suite)

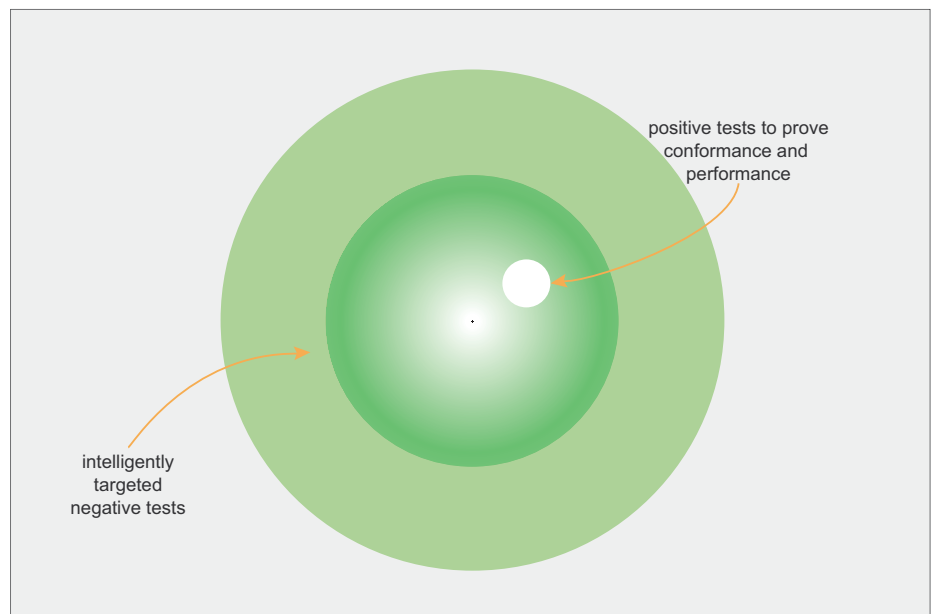


Figure 1. Infinity of negative inputs, and test optimization (source: Fuzzing for Software Security by Takanen et al., by Artech House, in press)

reaches the highest coverage result is not necessarily the best at finding security-critical flaws in the product. A good model can reach high code coverage even without a good anomaly library. This indicates that code coverage is very bad at measuring the goodness of negative testing. On the other hand, a random fuzzer will typically reach very low code coverage, indicating that it will not test anything but the simplest structures in the inputs. A presentation by Charlie Miller at CanSecWest revealed that the code coverage for intelligent model-based fuzzers could be more than two times higher than for the less intelligent fuzzers.

Coverage Through Interfaces and Attack Vectors

Test coverage can also be analyzed through the various attack vectors

Table 1. Code Coverage for OpenSSL (Source: Tero Rontti, Robustness Testing Code Coverage Analysis, Master's Thesis, 2004)

Metric/Test Suite	Model-based Fuzzer	Conformance Suite
Line Coverage	20.1%	19.7%
Branch Coverage	5.1%	4.2%
Function Coverage	17.4%	16.2%

and layers of protocols on each of those identified interfaces. The most common use case for fuzzing is in Web development. Several commercial tools from companies such as Cenzic, HP and IBM are available for testing various Web applications, some of which are quite intelligent in their approach to negative testing. However, for any communication device or network service, designers need to take a step back and see what other communication is happening. Almost any enterprise environment today depends on communication technologies such as HTTP and SSL/TLS (for Web services), SIP and RTP (for VoIP), ISAKMP, IKE and IPSEC (for VPN connectivity), and SMTP, POP and IMAP (for e-mail). Protocols like these are familiar to any security engineer that has configured a firewall for enterprise use and creating threat scenarios should be simply based on that knowledge. For any

network-enabled service, testers need to fuzz all layers. For example, for testing a VoIP service, they need to test IPv4, TLS, SIP, RTP, and the actual application logic including various voice encoding formats (Figure 3).

Input Space Coverage

The most challenging metric is related to measuring the actual inputs given to any of the above-mentioned interfaces. For any meaningful metrics the interface needs to have a formal description, which can also be deduced from sampled of network packets. For example, for a simple protocol such as TFTP, the interface description can be something like this (using BNF-like description, simplified for brevity):

```
<RRQ> ::= (0x00 0x01)
        <FILE-NAME> <MODE>
<WRQ> ::= (0x00 0x02)
        <FILE-NAME> <MODE>
<MODE> ::= ("octet" | "netascii") 0x00
<FILE-NAME> ::= { <CHARACTER> } 0x00
<CHARACTER> ::= 0x01 -0x7f
```

Measuring the input space would be based on analyzing each of the inputs

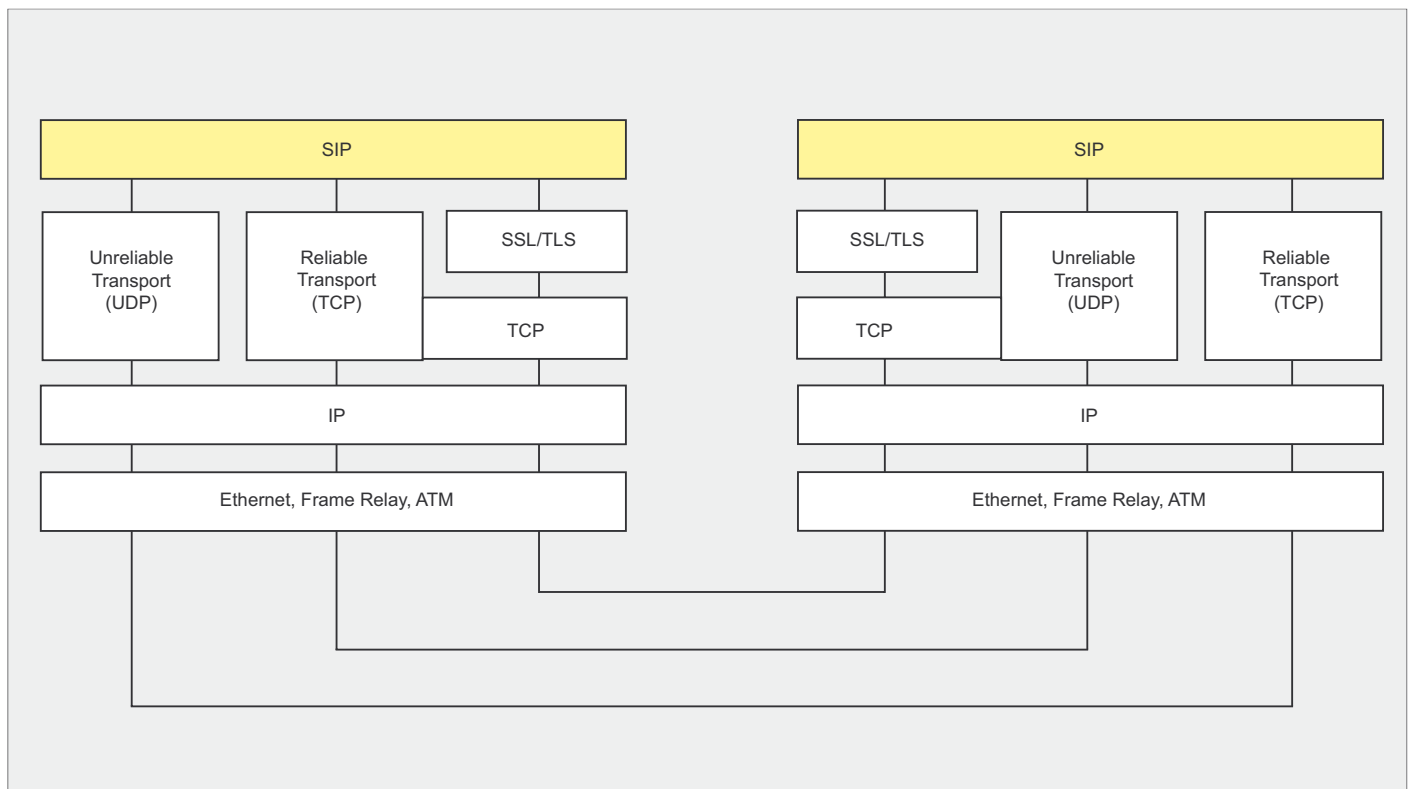


Figure 3. Example protocol stack from VoIP, showing various layers and interfaces that need to be tested (source: Securing VoIP Networks by Thermos and Takanen, Addison-Wesley 2007)

used for all elements inside the protocol messages. For example, trying only 10 inputs for a file name would result in a smaller input coverage than using 100 inputs. The problem with automated input space coverage is in analyzing how meaningful each of those inputs really were.

The Ultimate Goal – Finding a Flaw

Fuzzing is not only about generation of test cases. It is also about detecting and fixing the found issues. A security guy might be happy with one new zero-day mistake in the product and might not worry about the time it takes to actually fix the found flaws. The software developers, however, only care about the easiness of fixing the found issues. A problem that cannot be easily reproduced will not get a high priority in the long bug lists that they are working on.

The debugging phase starts when testers do find a crash-level flaw. Preparation for that outcome has to start immediately when planning tests. All commercial tools have various test control APIs that testers can use to restart the test target and to collect various metrics from the tests and the target system. Network analyzers can be used to collect all network traffic and enable reproduction of the tests through capture-replay tools, if such functionality is not included in the test tool itself. For debugging the actual flaw, it is important to know the various flaw categories in software. Catching the actual flaw from the code can be very difficult if you do not know what a buffer overflow looks like and how to fix it. Therefore, it is beneficial to do

training on secure programming skills for the entire development team.

Example Test Results

Last year, we at Codenomicon collected some research findings from testing various wireless devices and compiled a white paper on that topic using our commercial model-based fuzzers. The study, for example, revealed that fuzzing was able to break down 28 out of 31 Bluetooth devices we tested. Figure 4 shows similar experiences from testing wireless access points. Everything broke when fuzzing was deployed to WiFi devices (Figure 4). The resulting error modes ranged from crashing processes and services, to total reboot of the device, or even corruption of the internal flash memory in the embedded device requiring reprogramming of the device to fix it. Clearly, the industry is not yet deploying negative testing in all different software industries.

Conclusion

The main problem for fuzzing today is that too few people do it. For those who already use fuzzing, the greatest challenge appears to be metrics. If you cannot measure it, you cannot improve it, and you surely do not understand it. All types of metrics are critical for the deployment of useful fuzzing techniques.

The most important part of deploying negative testing is to conduct a good interface analysis and to understand what you need to fuzz. Through that analysis, you can map down the attack surface of the system under test. A limited fuzz test against one interface or against one layer on that interface is not enough to verify the

security of the system. A fuzzer tool alone is not enough for the security test. The test engineer or the security expert conducting a penetration test against the system has to understand the inner operations of the product. Integration to system monitoring tools and test controllers is crucial for a successful test.

Where do we see the usage of fuzzing today and tomorrow? IT decision makers at software companies should deploy negative testing because of the direct cost benefits and advantages associated with it. A flaw identified proactively before deployment has enormous value to them. Identifying specific protocol and permutations of inputs, software testers are able to determine difficult issues using this model-based optimized method. Additionally, they do not waste time trying to explore the infinite amount of inputs to determine the particular test that causes anomalous behavior. Negative testing solves this issue by allowing them to apply their previous experience with typical problem areas to target specific test cases, giving them more time to work on other higher-priority tasks. Negative testing also has benefits for the customer: code has fewer defects, so there is less public exposure to attacks and that makes for a better experience for the software end user. There are no false positives in fuzzing. Negative testing is just one piece of the puzzle of security testing techniques, and fuzzing is just one form of negative testing. But possibly it is the most cost effective form of security testing, depending how well you deploy it.

About the Authors

Mikko Varpiola is a Codenomicon founder and security/robustness solutions architect. Before founding the company, he was a key researcher at a globally recognized security testing research group, Oulu University Secure Programming Group (OUSPG). Mikko is the company's leading expert on hacking communication interfaces, and speaks more than 160 protocols fluently. He has been involved with the development of fuzzers for all those protocols and in using those tools to conduct the best security assessments in the world as part of Codenomicon's services offering.

Ari Takanen is a founder and CTO of Codenomicon Ltd., an accomplished speaker and published author on the topic of software security testing. In addition, Ari has an extensive background in the academic world through his software security testing research at PROTOS/OUSPG. During his research he developed his passion for identifying and eliminating the coding errors and software weaknesses that threaten businesses and individuals alike. He has also written two books on how to identify security weaknesses.

	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 4. Test results from testing various WiFi Access Points (source Codenomicon white paper 2008)