Fuzzing : the Past, the Present and the Future

Ari Takanen

Codenomicon Ltd., Tutkijantie 4E, Finland

Résumé Standards in communication enable new test automation techniques for verification of reliability and security. Fuzzing is a negative software testing method 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, and will focus on finding issues with invalid data elements, broken syntax of messages, but also unexpected message sequences. This talk will focus on the most recent advancements in model-based fuzzers, including the challenges in XML fuzzing. XML is a structure that is today used everywhere, from web applications to mission critical systems in SCADA and telecommunications. And unfortunately, based on research conducted at CodeLabs, XML is extremely error-prone due to its complexity.

1 Introduction

Fuzzing is a negative software testing method 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, implementations of industry standard protocols. The availability of extensive documentation for the interfaces means that fuzzing is able to cover the most exposed and critical attack surfaces in a system with systematic and optimized tests, and therefore identify many common errors and potential vulnerabilities quickly and cost-effectively. There are no false positives in fuzzing. Tests can be conducted against any system, whether it is internally built, or developed by third parties. Fuzzing is a black-box testing technique that does not require any access to the source code of the system under test. It can be used in any phase of the software life-cycle.

1.1 Use of Fuzzing

Fuzzing is relatively new test automation technique for finding critical security problems in communication software. Only a year ago, 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. In the BSIMM study [1], nine leading product security teams were interviewed, and all of them were noted to use fuzzing as part of the security engineering process. Commercial fuzzing tool vendors have similar findings : 80% of leading service providers and device manufacturers are using or are in process of deploying fuzzing. As awareness of best practises in product security spreads, and is educated openly, also smaller players in the software industry are slowly adapting fuzzing into their processes. Therefore, one can say that fuzzing is finally a mainstream testing technique used by all major companies building software and devices for critical communication infrastructure.

1.2 Brief History of Fuzzing

The history of fuzzing is quite long. The term “fuzzing” or “fuzz testing” emerged in 1988 [2], but in its original meaning fuzzing was just another name for random testing, with very little use in QA beyond some limited ad-hoc testing. Even before that, random testing was used to find reliability issues in communication software. The transition to integrating the approach into software development was evident but not very common outside few mission critical applications.

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 [3]. 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 [4].

Much later, around year 2007 also other companies became fascinated in the topic of protocol fuzzing, and a new highly competed test and measurement market domain was created. First open source fuzzing frameworks emerged around that same time. Finally in 2008, among other books on fuzzing, 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].

1.3 Overview of this Article

The purpose of this article is to give you an overview of fuzzing, so that you can understand other fuzzing related technologies easier, and can place them in the technology map around fuzzing. The goal is to enable you to understand how different fuzzing technologies work, and where to use them. This article will first explore the principles behind fuzzing, test automation and penetration testing. Then we will explore some specifics on how protocols are built, and how fuzzing will fit with each protocol type.

2 Fuzzing Overview

Any fuzzing is better than doing no fuzzing at all. To understand the principles behind fuzzing and the strengths in each technology, we need to look how fuzzing fits into the entire software lifecycle.

2.1 Requirements and Fuzzing

As the software development process starts from requirements gathering, so letÕs 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 as shown in Fig. 1 [6]. 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. Similarly, fatal features and unwanted features should be defined somewhere so that they will be tested for, and that is the purpose of negative requirements. The undefined 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 decisions 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 behavior of the software but to explore the challenging area of negative requirements.

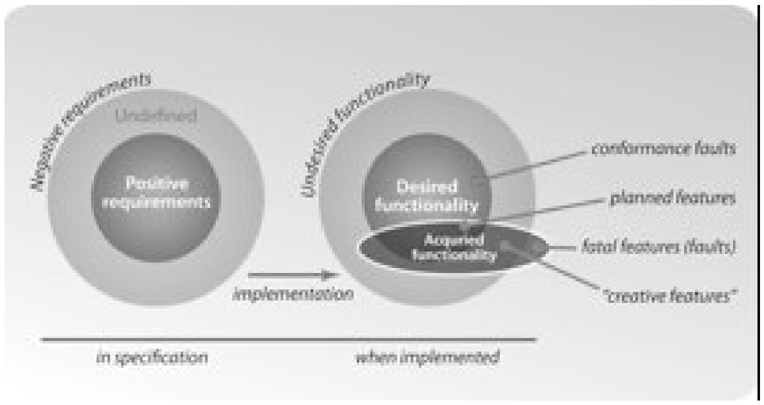

Fig. 1. Specification versus implementation

2.2 Black-box Testing Techniques

Looking at different types of black-box testing, we can identify three main categories of testing techniques. 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 test the infinite input space that is required to try out in robustness testing.

2.3 Purpose of Fuzzing

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 related to the value of a software vulnerability 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.

3 Test Automation Techniques

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. 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. 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.

3.1 Test Generation in Fuzzing

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 [7]. The resulting executable model is basically a full implementation of one of the end-points in the communication.

3.2 Fuzzing Adaptation Strategies

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 and 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.

3.3 Comparison of Fuzzers

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 a study by Charlie Miller [5][8], fuzzers were compared by running their tests against a piece of software that had intentionally planted security vulnerabilities in it. Based on that sample, the fuzzer efficiency was noted to range from 0% up to 80%, with random testing providing very inefficient test results and model based tests peaking at higher efficiency. 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. Code coverage metrics also indicated that best code coverage did not directly reveal the best fuzzer.

4 Use of Fuzzing in Penetration Testing and Security Audit

Fuzzing is a one technique for automating security audits. The terms for security related auditing and other forms of consulting vary from penetration testing to security certifications. What is a penetration test ? What does a security audit consist of ? What does a consultant mean when he talks about security assessment ? Clarification is needed, as there are no definite definitions for any of these.

4.1 Security Certifications

Security Certification has to be done against a (public, validated) specification. The specification can consist of an industry standard questionnaire or exam (such as personnel certification), requirement specification (as in common criteria), or can be similar to an industry standard quality audit (similar to ISO-9001). It is a public stamp of approval, and the quality of a certificate can depend on who actually conducted the audit.

4.2 Security Audit

Security Audit is the process of auditing a system. It can be proprietary (such as ICSA Labs) or industry standard (such as Common Criteria). Audit is the process of doing a certification. Depending on the goal, different tools are used. Most audits are more process oriented rather than technical, but that is only because technical audits (if specified carefully) easily turn into check-lists or tools. A security audit can very rarely contain ad-hoc or innovative auditing methods as those are often impossible to specify in form of requirements specifications. An extreme example of such attempt is the torture tests of IETF, or the robustness profile of Common Criteria.

4.3 Vulnerability Assessment

Vulnerability Assessment can be both reactive and proactive, and can contain penetration testing. Unfortunately 90% of security consultants rely on tools, so people identify this process with security auditing/scanning tools. These tools can be both black-box and white-box. Fuzzing tools can be part of this process. For example, PROTOS is already a standard tool for many vulnerability assessment people, due to it being freely available. Vulnerability assessment should not be called security audit unless it is a standard process, and aims for some kind of semi-certification against that proprietary or industry standard process.

4.4 Penetration Testing

Penetration Testing is the lowest level of testing, and is one of the practical assessment techniques used in many forms of security audits. The tools and methodologies in penetration testing are not restricted, and are often chosen based on availability and skills of the auditor. There is basically no clear definition of what is done in a penetration test. It is hands-on tiger-team testing. It can be very innovative and ad-hoc. But again, most consultants will use standard tools. Biggest difference to vulnerability assessment is that the purpose is to penetrate the system. A good penetration testing team consist of people with hacker mentality that understand what can go wrong with the product and can simulate those events with penetration tests to realize the compromise of the system. A vulnerability assessment can reveal a potential that there is an exploitable flaw (such as a bad system call) and the penetration testing people can create an attack against that. Penetration testing is much more narrow in focus than a full vulnerability assessment, and it is very difficult to do well, as it requires knowledge of protocols and systems. Fuzzing tools automate this process, building the expertise into software rather than having the tests manually built per assignment. The use of difficult to measure random fuzz testing is always restricted to penetration tests. Whereas systematic model-based robustness testing can also be adapted as a standard, being based on the communication interface specifications, and therefore being more measurable criteria.

5 Data Representation Formats

Formal descriptions are essential for describing protocols and interfaces. The more structural a protocol is, the more protocol awareness is required from a fuzzer.

5.1 ASCII

Most ASCII protocols, such as HTTP, are represented in simple text formats indicating name and value pairs. The example below shows a fuzzed HTTP response testing browser reliability using PROTOS HTTP Client [9] test suite (the test case is fetched using NetCat) :

```
$ echo "get /" | nc localhost 8000
HTTP/1.1 200 OK
Server: Bugbear/020
Date: Monday, 03-Jan-02 12:12:12 GMT
Vary: *
Connection: close
Last-Modified: Wed, 14 Jan 1970 06:56:40 EET
Content-Type: text/html; charset=ISO-8859-1
Content-Length: 80
Cache-Control: %s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s
```

Note that not all ASCII protocols use simple request-response sequences. For example SIP consists of complex message sequences and state transitions. The same principles for ASCII fuzzing still apply for all ASCII protocols.

Fuzzing is highy effective in catching simple buffer overflow and format string issues among many other more complex vulnerabilities by triggering them with real semi-hostile inputs. Most ASCII protocol fuzzers are simple mutation based fuzzers, as there is very little dependence across different name-value pairs. The most complex features of ascii protocols are related to calculating check-sums and hash-values.

5.2 ASN.1

Using a powerful library of encoding rules, ASN.1 protocols represent complex structures in compact binary protocols (example below is from Codenomicon X.509 fuzzer) :

```
X.509 Certificate:
 SEQUENCE:
   Type:
     UNIVERSAL: 0b00
     C: 0b1
     No-16: 0b10000
   Length: 04
   Value:
     TBSCertificate:
       SEQUENCE:
         Type:
           UNIVERSAL: 0b00
           C: 0b1
           No-16: 0b10000
         Length: 82015d
```

```
                Value:
                  version: DEFAULT: ()
                  CertificateSerialNumber: INTEGER:
                    Type:
                      UNIVERSAL: 0b00
                      P: 0b0
                      No-2: 0b00010
                    Length: 14
                    Value:   d7 00 67 27 30 7b c4 41 f9 4f 05 3a c3 5f 49 c4 46 67 83
                         aa
                   ...
```

Mutation-based fuzzers have very little chances of receiving any sensible outcome from fuzzing ASN.1 protocols. Actually most fuzzing frameworks will completely ignore support for binary protocols. Semi-random changes can still break the implementations, but most changes are detected on the parser level and will never be passed to higher level applications.

5.3 XML

XML protocols combine the hierarchical structures of ASN.1 with text representations. XMPP is an example communication protocol using XML structures :

```
      <?xml version='1.0'?>
      <stream:stream xmlns='jabber:client' \
      xmlns:stream='http://etherx.jabber.org/streams' \
      id='none' from='im.codenomicon.com' version='1.0'>
        <stream:error>
          <xml-not-well-formed xmlns='urn:ietf:params:xml:ns:xmpp-streams'/>
        </stream:error>
      </stream:stream>
```

The flexibility of XML structures enable XML fuzzers to explore deeply recursive tests, but also each element in the name-value pairs can contain complex structures that are passed as-is all up to the user interface components. Random testing can easily deduct simple structures, but most changes are detected by the XML parsers, and with some empiric XML fuzzing excercises people have noted that less than 2% of the tests actually reach higher level applications. Similarly as in ASN.1 fuzzing, this can be impacted by bringing more protocol awareness into the test generation process.

6 Conclusions

Fuzzing is here to stay, and if you will not use it yourself, someone else will do it for you. Fuzzing tools are easily available as well supported commercial solutions but also as free open source solutions, so not doing it 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 other research teams are embracing fuzzing as a new security research topic.

Notes and Comments. In this article, we provided some background to fuzzing, and the presentation will go deeper into actual use cases of fuzzing. Note also, that these same topics are covered more extensively in [5]. This article is loosely based on that book.

Références

1. Gary McGraw, Brian Chess, Sammy Migues : Building Security In Maturity Model (BSIMM). Online publication. Available : `http://bsi-mm.com/`
2. Barton Miller et al : The Fuzz Project. Related publications available at : `//pages.cs.wisc.edu/~bart/fuzz/fuzz.html`
3. OUSPG Research Group : PROTOS Security Testing of Protocol Implementations. Related publications available at : `//www.ee.oulu.fi/research/ouspg/protos/`
4. Codenomicon Ltd. `//www.codenomicon.com/`
5. Ari Takanen, Jared DeMott, Charlie Miller : Fuzzing for Software Security Testing and Quality Assurance. Published by Artech House, 2008.
6. Juhani Eronen, Marko Laakso : A Case for Protocol Dependency. In Proceedings of the First IEEE International Workshop on Critical Infrastructure Protection. Darmstadt, Germany. November 3-4, 2005.
7. Rauli Kaksonen : A Functional Method for Assessing Protocol Implementation Security. Licentiate thesis. Espoo. 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.).
8. Charlie Miller : Fuzz By Number. Presentation at CanSecWest 2008. Available at : `//cansecwest.com/csw08/csw08-miller.pdf`
9. PROTOS Test-Suite : c05-http-reply. Available at : `//www.ee.oulu.fi/research/ouspg/protos/testing/c05/http-reply/`