Review on Fuzzer Project

MCSH2493-01 PEMBANGUNAN PERISIAN SELAMAT (SECURE SOFTWARE DEVELOPMENT)

Mohd Azhari bin Kamarudin (MCS171006)

Nordziakon bin Husnan (MCS171001)

Nazrin

Contents

1.Introduction	
2.History	
3.Fuzzer implementations	
4.Comparison with cryptanalysis	
5.Application of fuzzing	
6.Fuzzing tools	
7.Protocol of fuzzing	
8.Fuzzers advantages and limitation	
9.References	

1.Introduction

Fuzzing is a effective software technique for getting bugs detected . The idea is very simple: create a lot of randomly malformed inputs to decode a program and see what's going on. If the system crashes then possibly something is wrong. Although fuzzing is a well-known technique, it's shockingly easy to find vulnerabilities in commonly used apps, even with security implications.

Fuzz testing or Fuzzing is a Black Box software testing technique which basically consists of finding implementing bugs using automated injection of malformed / semi-malformed data. In the world of cybersecurity, fuzzing is the usually automated process of finding hackable software bugs by randomly feeding different permutations of data into a target program until one of those permutations reveals a vulnerability. It's an old but increasingly common process both for hackers seeking vulnerabilities to exploit and defenders trying to find them first to fix. And in an era when anyone can spin up powerful computing resources to bombard a victim application with junk data in search of a bug, it's become an essential front in the zero-day arms race.

2.History

In 1989 Professor Barton Miller and his students developed Fuzz Testing at the University of Wisconsin Madison is primarily based on command-line and UI fuzzing, and demonstrates that current operating systems are vulnerable to even pure blurring.The term "fuzzing" originates from a 1988 class project, taught by Barton Miller at the University of Wisconsin.(Ari Takanen,2018) To fuzz test a Unix utility meant to automatically generate random files and command-line parameters for the utility. The project was designed to test the reliability of Unix programs by executing a large number of random inputs in quick succession until they crashed. It also provided early debugging tools to determine the cause and category of each detected failure. To allow other researchers to conduct similar experiments with other software, the source code of the tools, the test procedures, and the raw result data were made publicly available(University of Wisconsin-Madison, Retrieved 2009) Later, the term fuzzing was not limited only to command-line utilities.

3.Fuzzer implementations

A fuzzer is a program which injects automatically semi-random data into a program/stack and detect bugs.The data-generation part is made of generators, and vulnerability identification relies on debugging tools. Generators usually use combinations of static fuzzing vectors (known-to-be-dangerous values), or totally random data. New generation fuzzers use genetic algorithms to link injected data and observed impact. Such tools are not public yet.

A fuzzer can be categorized in several ways:

1. A fuzzer can be generation-based or mutation-based depending on whether inputs are generated from scratch or by modifying existing inputs.
2. A fuzzer can be dumb or smart depending on whether it is aware of input structure.
3. A fuzzer can be white-, grey-, or black-box, depending on whether it is aware of program structure.

4.Comparison with cryptanalysis

The number of possible tryable solutions is the explorable solutions space. The aim of cryptanalysis is to reduce this space, which means finding a way of having less keys to try than pure bruteforce to decrypt something.

Most of the fuzzers are:

- protocol/file-format dependant
- data-type dependant

The reason is

- First, because the fuzzer has to connect to the input channel, which is bound to the target.
- Second, because a program only understands structured-enough data. If you connect to a web server in a raw way, it will only respond to listed commands such as GET (or eventually crash). It will take less time to start the string with “GET “, and fuzz the rest, but the drawback is that you’ll skip all the tests on the first verb.

In this regard, Fuzzers try to reduce the number of unuseful tests, i.e. the values we already know that there’s little chance they’ll work: you reduce unpredictability, in favor of speed.

Attack types

A fuzzer would try combinations of attacks on:

- numbers (signed/unsigned integers/float…)
- chars (urls, command-line inputs)
- metadata : user-input text (id3 tag)
- pure binary sequences

A common approach to fuzzing is to define lists of “known-to-be-dangerous values” (fuzz vectors) for each type, and to inject them or recombinations.

- for integers: zero, possibly negative or very big numbers
- for chars: escaped, interpretable characters / instructions (ex: For SQL Requests, quotes / commands…)
- for binary: random ones

Protocols and file formats imply norms, which are sometimes blurry, very complicated or badly implemented : that’s why developers sometimes mess up in the implementation process (because of time/cost constraints). That’s why it can be interesting to take the opposite approach: take a norm, look at all mandatory features and constraints, and try all of them; forbidden/reserved values, linked parameters, field sizes. That would be conformance testing oriented fuzzing.

5.Application of fuzzing

Whatever the fuzzed system is, the attack vectors are within its I/O. For a desktop app:

- the UI (testing all the buttons sequences / text inputs)
- the command-line options
- the import/export capabilities (see file format fuzzing below)

For a web app: urls, forms, user-generated content, RPC requests, …

6.Fuzzing tools

These are some of the available tool for fuzzing , which is

Open Source

Mutational Fuzzers

1. american fuzzy lop
2. Radamsa - a flock of fuzzers

Fuzzing Frameworks

1. Sulley Fuzzing Framework
2. boofuzz
3. BFuzz

Domain-Specific Fuzzers

1. Microsoft SDL MiniFuzz File Fuzzer
2. Microsoft SDL Regex Fuzzer
3. ABNF Fuzzer

Commercial products

1. Codenomicon's product suite
2. Peach Fuzzing Platform
3. Spirent Avalanche NEXT
4. Beyond Security's beSTORM product

Deprecated Tools

Untidy - XML Fuzzer (Now integrated into Peach)

7.Protocol of fuzzing

A protocol fuzzer sends forged packets to the tested application, or eventually acts as a proxy, modifying requests on the fly and replaying them.

File format fuzzing

A file format fuzzer generates multiple malformed samples, and opens them sequentially. When the program crashes, debug information is kept for further investigation.

One can attack:

- the parser layer (container layer): file format constraints, structure, conventions, field sizes, flags, …

- the codec/application layer: lower-level attacks, aiming at the program's deeper internals

One example of file format related vulnerabilities: MS04-028 (KB833987) Microsoft's JPEG GDI+ vulnerability was a zero sized comment field, without content.

Surprisingly, file format fuzzers are not that common, but tend to appear these days; some examples:

- A generic file format fuzzer : Ilja van Sprundel's mangle.c; "it's usage is very simple, it takes a filename and headersize as input. it will then change approximatly between 0 and 10% of the header with random bytes." (from the author)

- Zzuf can act as a fuzzed file generator, http://sam.zoy.org/zzuf/

- One may use tools like Hachoir as a generic parser for file format fuzzer development.

8.Fuzzers advantages and limitation

The great advantage of fuzz testing is that the test design is extremely simple, and free of preconceptions about system behavior

The systematical/random approach allows this method to find bugs that would have often been missed by human eyes. Plus, when the tested system is totally closed (say, a SIP phone), fuzzing is one of the only means of reviewing it's quality.

Fuzzers limitations

Fuzzers usually tend to find simple bugs; plus, the more a fuzzer is protocol-aware, the less weird errors it will find. This is why the exhaustive / random approach is still popular among the fuzzing community.

Another problem is that when you do some black-box-testing, you usually attack a closed system, which increases difficulty to evaluate the dangerosity/impact of the found vulnerability (no debugging possibilities).

Fuzzing advantages

The purpose of fuzzing relies on the assumption that there are bugs within every program, which are waiting to be discovered. Therefore, a systematical approach should find them sooner or later.

Fuzzing can add another point of view to classical software testing techniques (hand code review, debugging) because of its non-human approach. It doesn't replace them, but is a reasonable complement, thanks to the limited work needed to put the procedure in place.

Fuzzing has grown from a low-budget technique used by individual hackers to a kind of table-stakes security audit performed by major companies on their own code. Lone hackers can use services like Amazon to spin up armies of hundreds of computers that fuzz-test a program in parallel. And now companies like Google also devote their own significant server resources to throwing random code at programs to find their flaws, most recently using machine learning to refine the process. Companies like Peach Fuzzer and Codenomicon have even built businesses around the process.

Example of fuzzing initiatives:

- The Month of Kernel Bugs, which revealed an Apple Wireless flaw
  tools

- The Month of Browser Bugs (review here); number of bugs found: MSIE: 25 Apple Safari: 2 Mozilla: 2 Opera: 1 Konqueror: 1; used DHTML, Css, DOM, ActiveX fuzzing tools

- Fuzzing with WebScarab: a framework for analysing applications that communicate using the HTTP and HTTPS protocols

- JBroFuzz: a web application fuzzer

- WSFuzzer: real-world manual SOAP pen testing tool

9.References

John Neystadt (February 2008). "Automated Penetration Testing with White-Box Fuzzing". Microsoft. Retrieved 2009-05-14.

Gerald M. Weinberg (2017-02-05). "Fuzz Testing and Fuzz History". Retrieved 2017-02-06.

Joe W. Duran; Simeon C. Ntafos (1981-03-09). A report on random testing. Icse '81. Proceedings of the ACM SIGSOFT International Conference on Software Engineering (ICSE'81). pp. 179–183. ISBN 9780897911467.

Joe W. Duran; Simeon C. Ntafos (1984-07-01). "An Evaluation of Random Testing". IEEE Transactions on Software Engineering. IEEE Transactions on Software Engineering (TSE) (4): 438–444. doi:10.1109/TSE.1984.5010257.

"Macintosh Stories: Monkey Lives". Folklore.org. 1999-02-22. Retrieved 2010-05-28.

Jump up to:[a] [b] [c] *Ari Takanen; Jared D. Demott; Charles Miller (31 January 2018). Fuzzing for Software Security Testing and Quality Assurance, Second Edition. Artech House. p. 15. ISBN 978-1-63081-519-6.* full document available (archived September 19, 2018)

"Fuzz Testing of Application Reliability". University of Wisconsin-Madison. Retrieved 2009-05-14.

"crashme". CodePlex. Retrieved 2012-06-26.

Jump up to:[a] [b] *Michael Sutton; Adam Greene; Pedram Amini (2007). Fuzzing: Brute Force Vulnerability Discovery. Addison-Wesley. ISBN 978-0-321-44611-4.*

Jump up to:[a] [b] *"Announcing ClusterFuzz". Retrieved 2017-03-09.*

Perlroth, Nicole (25 September 2014). "Security Experts Expect 'Shellshock' Software Bug in Bash to Be Significant". The New York Times. Retrieved 25 September 2014.

Zalewski, Michał (1 October 2014). "Bash bug: the other two RCEs, or how we chipped away at the original fix (CVE-2014-6277 and '78)". lcamtuf's blog. Retrieved 13 March 2017.

Seltzer, Larry (29 September 2014). "Shellshock makes Heartbleed look insignificant". ZDNet. Retrieved 29 September2014.

Böck, Hanno. "Fuzzing: Wie man Heartbleed hätte finden können (in German)". Golem.de (in German). Retrieved 13 March 2017.

Böck, Hanno. "How Heartbleed could've been found (in English)". Hanno's blog. Retrieved 13 March 2017.

"Search engine for the internet of things – devices still vulnerable to Heartbleed". shodan.io. Retrieved 13 March 2017.

"Heartbleed Report (2017-01)". shodan.io. Retrieved 10 July2017.

Walker, Michael. "DARPA Cyber Grand Challenge". darpa.mil. Retrieved 12 March 2017.

"Mayhem comes in first place at CGC". Retrieved 12 March2017.

Jump up to:[a] [b] *"Announcing Project Springfield". 2016-09-26. Retrieved 2017-03-08.*

Jump up to:[a] [b] [c] [d] *"Announcing OSS-Fuzz". Retrieved 2017-03-08.*

Christopher Domas (August 2018). "GOD MODE UNLOCKED - Hardware Backdoors in x86 CPUs". Retrieved 2018-09-03.

Offutt, Jeff; Xu, Wuzhi (2004). "Generating Test Cases for Web Services Using Data Perturbation". Workshop on Testing, Analysis and Verification of Web Services.

Rebert, Alexandre; Cha, Sang Kil; Avgerinos, Thanassis; Foote, Jonathan; Warren, David; Grieco, Gustavo; Brumley, David (2014). "Optimizing Seed Selection for Fuzzing" (PDF). Proceedings of the 23rd USENIX Conference on Security Symposium: 861–875.

Jump up to:[a] [b] [c] Patrice Godefroid; Adam Kiezun; Michael Y. Levin. "Grammar-based Whitebox Fuzzing" (PDF). Microsoft Research.

Jump up to:[a] [b] [c] Van-Thuan Pham; Marcel Böhme; Abhik Roychoudhury (2016-09-07). "Model-based whitebox fuzzing for program binaries". Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering - ASE 2016. Proceedings of Automated Software Engineering (ASE'16). pp. 543–553. doi:10.1145/2970276.2970316. ISBN 9781450338455.

Jump up to:[a] [b] [c] "Peach Fuzzer". Retrieved 2017-03-08.

Greg Banks; Marco Cova; Viktoria Felmetsger; Kevin Almeroth; Richard Kemmerer; Giovanni Vigna. SNOOZE: Toward a Stateful NetwOrk prOtocol fuzZEr. Proceedings of the Information Security Conference (ISC'06).

Osbert Bastani; Rahul Sharma; Alex Aiken; Percy Liang (June 2017). Synthesizing Program Input Grammars. Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017). arXiv:1608.01723. Bibcode:2016arXiv160801723B.

Wang, T.; Wei, T.; Gu, G.; Zou, W. (May 2010). TaintScope: A Checksum-Aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection. 2010 IEEE Symposium on Security and Privacy. pp. 497–512. CiteSeerX 10.1.1.169.7866. doi:10.1109/SP.2010.37. ISBN 978-1-4244-6894-2.

Patrice Godefroid; Michael Y. Levin; David Molnar (2008-02-08). "Automated Whitebox Fuzz Testing" (PDF). Proceedings of Network and Distributed Systems Symposium (NDSS'08).

Marcel Böhme; Soumya Paul (2015-10-05). "A Probabilistic Analysis of the Efficiency of Automated Software Testing". IEEE Transactions on Software Engineering. **42** (4): 345–360. doi:10.1109/TSE.2015.2487274.

Nick Stephens; John Grosen; Christopher Salls; Andrew Dutcher; Ruoyu Wang; Jacopo Corbetta; Yan Shoshitaishvili; Christopher Kruegel; Giovanni Vigna (2016-02-24). Driller: Augmenting. Fuzzing Through Selective Symbolic Execution(PDF). Proceedings of Network and Distributed Systems Symposium (NDSS'16).

Marcel Böhme; Van-Thuan Pham; Abhik Roychoudhury (2016-10-28). "Coverage-based Greybox Fuzzing as Markov Chain". Coverage-based Greybox Fuzzing as a Markov Chain. Proceedings of the ACM Conference on Computer and Communications Security (CCS'16). pp. 1032–1043. doi:10.1145/2976749.2978428. ISBN 9781450341394.

Hamlet, Richard G.; Taylor, Ross (December 1990). "Partition testing does not inspire confidence". IEEE Transactions on Software Engineering. **16** (12): 1402–1411.