Methods of generating test data for carrying out the fuzzing process

M. PACHNIK
marcin.pachnik@wat.edu.pl

Military University of Technology
ul. Kaliskiego 2, 01-489 Warsaw, Poland

The article presents and compares modern methods of generating test data in the process of automatic software security testing, so called fuzz testing. The publication contains descriptions of methods used, among others, in local, network or web applications, and then compares them and evaluates their effectiveness in the process of ensuring software security. The impact of the quality of test data corpus on the effectiveness of automated security testing has been assessed.

Keywords: fuzzing, test data corpus, security vulnerabilities.

DOI: 10.5604/01.3001.0013.6603

1. Introduction

Fuzzing is an automatic, pseudo-random software testing method. It consists in entering random, modified or erroneous test cases at the input of program in order to find an unhandled error [1]. This data is generated from scratch or on the basis of collected sets of correct test data, the so-called corpus. Fuzz tests were developed at the University of Wisconsin Madison in 1989 by Barton P. Miller, Louis Fredriksen and Bryan So. The aim of this project was originally to detect command line errors and to test the user interface of operating systems [2]. Fuzzing does not search for logical errors, but for those related to incorrect or dangerous use of the programming language by the application author. Nowadays, fuzzing is most often used for security audits of software written in languages such as C/C++ that do not have sufficient security mechanisms. The universality of this method also enables to test hardware solutions or operating systems [3]. The implementation quality of cryptographic algorithms has also been successfully tested [4].

Typical errors found with fuzz test programs (fuzzers) are memory protection errors (when the program accesses a memory area not allocated to it), buffer overflow, stack overflow, or variable overflow (when the size of structure exceeds the amount of memory allocated to it) [5]. These errors allow unauthorized access to program memory areas against the author's intentions.

Fuzzing can be divided according to the knowledge level of the source code and the way test cases are generated. Black-box fuzzing is characterized by a lack of knowledge of the source code by the tester, based on the ability to solely evaluate the correctness of the program execution. Grey-box fuzzing are similar to black-box fuzz tests. They are characterized by the knowledge of some partial information about the program under study through its analysis. White-box fuzzing require the tester knows the program code, detailed information about the program behavior during its execution and the ability to current analysis of the source code.

2. Selected test case generation methods

An important part of fuzz tests is to generate test data. Appropriate selection of the method significantly affects the effectiveness of tests. The algorithm cannot create test sets that would be rejected by the tested program. At the same time, it is advisable to generate as many tests sets as possible, the servicing of which was not implemented in analyzed program [6].

Pseudo-random method – pseudo-random generation consists in creating completely random data and providing the tested application with input values. This solution is useful in verification of used in the program under study validation method and correct implementation of data storage in memory (e.g. when the generated data is too large). However, it is characterized by a large number of rejected test cases, which reduces the effectiveness of fuzzing [1]. Figure 1 shows the construction of a fuzzer based on a pseudorandom generator. Random data generated by the generator is entered into a tested application. The application is monitored by an additional module that expects an unhandled error.

The errors found are deduplicated, resulting in an established set of found errors.

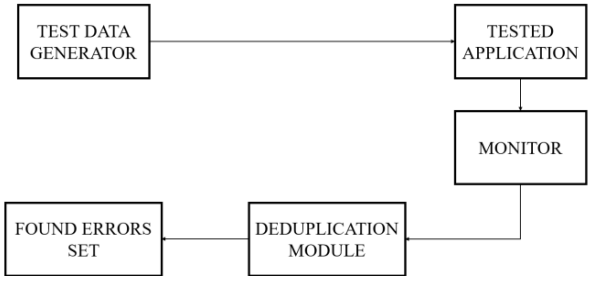

Fig. 1. Diagram of fuzzer based on a pseudo-random generator

Mutation method – based on a set of test cases – corpus. These cases may be published by the authors of tested software or collected by a person conducting the tests, e.g. using the Internet. They may also be selected test cases that have been chosen during previous tests. They may be characterized by a large code coverage of tested application or that they caused an error of another previously tested software or a previous version of the currently tested program. These cases are modified randomly (to a small extent), without taking into account the correctness of their structure [7]. Figure 2 presents the construction of a mutation fuzzer. The data entered into the tested program are modified test cases loaded from the corpus.

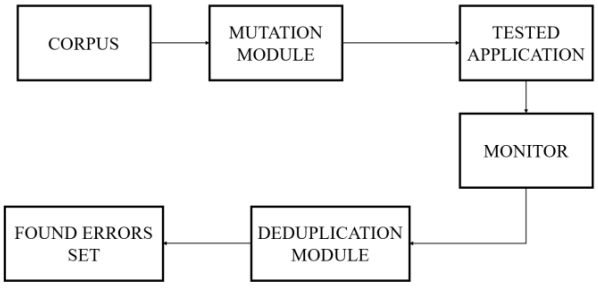

Fig. 2. Diagram of fuzzer generating test cases by mutation method

Grammatical method – in contrast to the mutation method takes into account the structure of test data. This method modifies test cases so that the newly created test case is correct in terms of file formatting. It can also be based on a corpus of test cases or, as in the case of pseudo-random method, generate completely new but structurally correct data [7]. Figure 3 presents the construction of a fuzzer based on a grammatical mutator. The principle of its operation is the same as in the case of a mutation fuzzer, except that random changes are made taking into account the correct structure of the file format of test case.

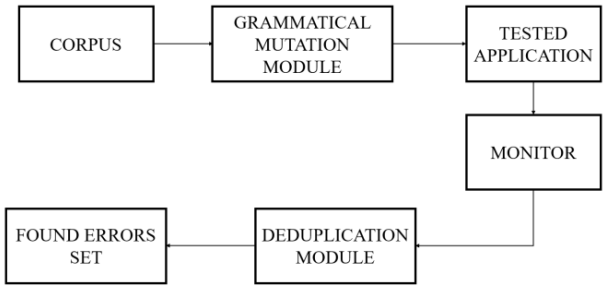

Fig. 3. Diagram of fuzzer generating test cases by grammatical method

Evolutionary method – uses the instrumentalization of tested software. On the basis of selected parameters, the program evaluates generated test cases in terms of their further usefulness and selects them [8]. Currently the most popular fuzzer based on this method is the American Fuzzy Loop by Michał Zalewski [9]. The AFL uses an author's algorithm working according to the following steps list:

1. Loading initial test cases.
2. Load the first/following case from the list.
3. Trim the file to the smallest size that does not change the program behavior.
4. Performing a mutation according to the selected classical (mutation) strategy.
5. If any of the new cases increases the coverage, add it to the list.

Figure 4 shows the scheme of an advanced evolutionary fuzzer. It is enriched with a module called the corpus manager. He is responsible for the sections of generated test cases and adding them to the corpus.

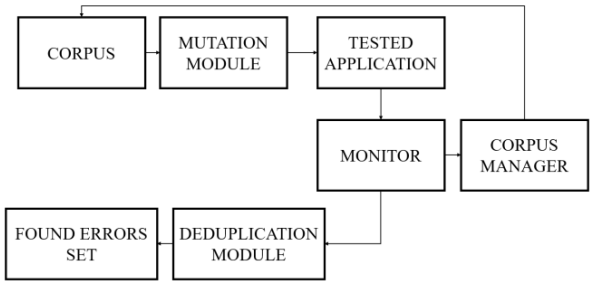

Fig. 4. Diagram of fuzzer generating test cases with an evolutionary algorithm

3. Technology used in tests

All the previously described methods (pseudo-random, mutation, grammatical and evolutionary method) have been implemented using the Python 3.5 language. Currently, the most popular fuzzers are written in lower-level languages such as

C/C++. This is due to their speed in relation to object-oriented or scripting languages [10]. For prototype purposes, Python allows faster project implementation in order to identify those methods that are worth further development and improvement.

The pseudo-random numbers in each of these algorithms are generated by the Mersene Twister algorithm (MT19937). It is an algorithm for generating pseudo-random numbers developed by M. Matsumoto and T. Nishimura [11] in 1997. Although it is not recommended for cryptographic solutions, this generator is fast and provides high quality pseudo-random numbers. It is characterized by a period of length 219937 and a high degree of distribution uniformity.

Code coverage can be measured by code line coverage, the number of code blocks activated by a test case or the number of activated edges in the control flow diagram. During the overview, the application code was covered according to the last criterion, analogous to the AFL fuzzer described above. For to measure the edge coverage, each critical edge in the program is divided into two by a "dummy" block (that process is shown on figure 5). In this way, it is possible to determine which path in the control flow graph activated a test case by instrumentalizing the program for activated code blocks.

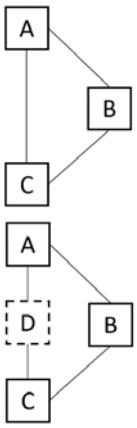

Fig. 5. Diagram showing the instrumentation of critical edge "A–C" by adding a block "D"

Tested software was compiled using clang compiler which uses *LLVM* (back-end) to generate, optimize code and to measure the edge coverage [12]. Clang configuration parameters for evolutionary method:

- *clang -std=c99 -g undertest.c -sanitize=address -fsanitize-coverage = trace-pc-guard,edge,no-prune*

enable to measure the code coverage. For other methods, the gcc compiler is used:

- *gcc undertest.c -std=c99*

In the second case, the GNU Debugger (GDB) was used to debug the program in case of a program failure.

The corpus used in tests was obtained from AFL project page [13]. Test suites composed of various types of graphic files dedicated to fuzzing process have been made available on this website.

During the overview, 1000 of the largest JPEG files available at the above-mentioned web address were used. In cases where the number of files in the corpus was reduced to one hundred and one element in succession, the files were randomly selected.

4. Tested software and comparison method

The security tests were conducted on a specially written sample program that performs operations on the loaded *.jpeg graphic file. It was written to contain three security vulnerabilities such as *integer overflow* and *buffer overflow*. The first error is caused by incorrect estimation of the value that a variable of the integer type can accept while the program is running. A possible consequence of such an error is overwriting the memory preceding the buffer or overflowing the buffer. Buffer overflow is a programming error consisting in writing to a designated memory area more data than the programmer reserved for this purpose. This situation leads to overwriting the data stored in the memory behind the buffer, which leads to erroneous operation of the program. Using this error, the invader can trigger specific actions. Each of the described methods was used during 24-hour tests. In order to carry them out (without the pseudo-random method, where each case is generated from scratch) three data corpus were used – consisting of 1, 100 and 1000 elements. The number of tests performed, the number of time-outs and the number of errors found, including unique errors, are interpreted as the experiment result.

5. Results

Test results were shown in tables 1–11.

Tab. 1. Number of errors found with the corpus size 1

Method/Corpus size	1		
Test time [h]	1	8	24
Mutation	293	2261	6679
Pseudo-random	622	5059	15684
Grammatical	1195	9628	29487
Evolutionary	1231	3826	12339

Tab. 2. Number of errors found with the corpus size 100

Method/Corpus size	100		
Test time [h]	1	8	24
Mutation	10	90	267
Pseudo-random	622	5059	15684
Grammatical	108	565	1636
Evolutionary	1518	38392	122880

Tab. 3. Number of errors found with the corpus size 1000

Method/Corpus size	1000		
Test time [h]	1	8	24
Mutation	39	354	1074
Pseudo-random	622	5059	15684
Grammatical	110	778	2348
Evolutionary	1399	36946	112038

Deduplication was carried out by comparison of function call stack, which in case of error detection by the debugger was saved with the test case calling it. This enabled the identification of where the error occurred in the tested program and consequently excluded recurring errors.

Tab. 4. Number of unique errors found with the corpus size 1

Method/Corpus size	1		
Test time [h]	1	8	24
Mutation	1	1	1
Pseudo-random	1	1	1
Grammatical	1	2	2
Evolutionary	1	2	2

Tab. 5. Number of unique errors found with the corpus size 100

Method/Corpus size	100		
Test time [h]	1	8	24
Mutation	1	1	2
Pseudo-random	1	1	1
Grammatical	1	2	2
Evolutionary	2	3	3

Tab. 6. Number of unique errors found with the corpus size 1000

Method/Corpus size	1000		
Test time [h]	1	8	24
Mutation	1	1	2
Pseudo-random	1	1	1
Grammatical	1	2	2
Evolutionary	2	3	3

The number of tests was influenced by the number of generated test cases, which caused the application to hang up. This resulted in a decrease in the number of tests, which translated into the number of errors found. During the tests, the maximum waiting time for the fuzzer to respond to the program was set at 1 second.

Tab. 7. Percentage share of generated test cases, which caused time-out for the corpus size 1

Method/Corpus size	1		
Test time [h]	1	8	24
Mutation	14%	13%	13%
Pseudo-random	20%	22%	21%
Grammatical	6%	7%	6%
Evolutionary	3%	5%	5%

Tab. 8. Percentage share of generated test cases, which caused time-out for the corpus size 100

Method/Corpus size	100		
Test time [h]	1	8	24
Mutation	40%	45%	44%
Pseudo-random	20%	22%	21%
Grammatical	16%	17%	16%
Evolutionary	1%	1%	1%

Tab. 9. Percentage share of generated test cases, which caused time-out for the corpus size 1000

Method/Corpus size	1000		
Test time [h]	1	8	24
Mutation	31%	33%	34%
Pseudo-random	20%	22%	21%
Grammatical	16%	17%	16%
Evolutionary	2%	2%	1%

As a result of the evolutionary algorithm, the corpus size of test cases has been modified, as shown in Table 3.

Tab. 10. Changes in the corpus size after fuzzing process with the use of AFL evolutionary algorithm

Corpus size	1	100	1000
The size of corpus	2	103	1006

Tab. 11. Changes in the corpus size after fuzzing process with the use of AFL evolutionary algorithm

Corpus size	1	100	1000
Size before tests	4,01 KB	81,1 KB	2398,8 KB
Size after tests	8,01 KB	83,0 KB	2410,3 KB

All the compared methods made it possible to find the security errors. Advanced methods are characterized by a smaller number of time-outs

and a greater number of detected unique errors. The use of the evolutionary method allowed enriching the corpus with the new test cases.

6. Conclusions

The highest rate of error detection was found for the method using an evolutionary algorithm, derived from the AFL program. It found all three vulnerabilities in the software. During the algorithm's operation, the corpus of test cases was modified. This allowed a significant increase in the number of tests performed. It was influenced by the number of time-outs caused by the tested program. The use of evolutionary algorithm enabled to eliminate test cases that hung up the tested application. This increased the number of tests performed and allowed for a more thorough analysis of the software. Considering the structure of peaks in the grammatical method also resulted in a reduction in the number of situations in which the tested software failed to respond. The comparison results indicate the important role of data corpus in the fuzzing process. A simple random selection of data is not effective due to undifferentiated test results. Too extensive corpuses cause a large number of time-outs, which significantly affects the number of tests performed. Sets of test cases that are too small cannot find a satisfactory number of security vulnerabilities.

7. Bibliography

[1] Takanen A., Demott J.D., Miller C., *Fuzzing for Software Security Testing and Quality Assurance*, Artech House, Norwood, 2008.

[2] So B., Fredriksen L., Miller B.P., "An empirical study of the reliability of UNIX utilities", *Communications of the ACM*, Vol. 33, 32–44 (1990).

[3] Schumilo S., Aschermann C., Gawlik R., Schinzel S., Thorsten H., "kAFL: Hardware--assisted feedback fuzzing for OS kernels", in: *Proceedings of the 26th USENIX Security Symposium*, pp. 167–182, 26th USENIX Security Symposium, Vancouver, BC, Canada, August 16–18, 2017.

[4] Böck H., *How Heartbleed could've been found* 2015, https://blog.hboeck.de/archives/868-How-Heartbleed-couldve-been-found.html.

[5] Josef Nelißen, *Buffer Overflows for Dummies*, SANS Institute, Swansea, 2002.

[6] Li J., Zhao B., Zhang Ch., "Fuzzing: a survey", *Cybersecurity*, Vol. 1, 1–13, (2018).

[7] Zeller A., Gopinath R., Böhme M., Fraser G., Holler Ch., *Generating Software Tests*, *https://www.fuzzingbook.org/*.

[8] Veggalam S., Rawat S., Haller I., Bos H., "Ifuzzer: An evolutionary interpreter fuzzer using genetic programming", *Lecture Notes in Computer Science*, Vol. 9878, 581–601 (2016).

[9] Zalewski M., *Technical "whitepaper" for afl-fuzz*, http://lcamtuf.coredump.cx/afl/technical_details.txt.

[10] Software in the Public Interest, Inc. – Debian Project *C++ g++ versus Python 3 fastest programs*, https://benchmarksgame-team.pages.debian.net/benchmarksgame/fastest/python3-gcc.html.

[11] Matsumoto M., Takuji Nishimura, "Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator", *ACM Transactions on Modeling and Computer Simulation*, Vol. 8, No. 1, 3–30 (1998).

[12] Lattner C., Adve V., "LLVM: a Compilation Framework for Lifelong Program Analysis & Transformation", in: *Proceedings of the International Symposium on Code Generation and Optimization*, Palo Alto California, USA, March 21–24, 2004.

[13] Zalewski M., *afl-generated, minimized image test sets*, http://lcamtuf.coredump.cx/afl/demo.

Metody generowania danych testowych służących do przeprowadzania „fuzz testing”

M. PACHNIK

W artykule przedstawiono i porównano współczesne metody generowania danych testowych w procesie automatycznego testowania bezpieczeństwa oprogramowania, tzw. *fuzz testing*. W publikacji zawarto opisy metod stosowanych m.in. w aplikacjach lokalnych, sieciowych czy webowych, a następnie dokonano ich porównania i oceny skuteczności w procesie zapewniania bezpieczeństwa oprogramowania. Oceniony został wpływ jakości korpusu (zbioru) danych testowych na efektywność przeprowadzania zautomatyzowanych testów bezpieczeństwa.

Słowa kluczowe: fuzzing, korpus danych testowych, błędy bezpieczeństwa.