Fuzzing

Daniel Basáez S.
dbasaez@inf.utfsm.cl
dbasaez@stud.fh-offenburg.de

January 22, 2009

Abstract

Fuzzing is a technique for Testing, and is very effective for finding security vulnerabilities in software. It was used in several applications like Unix systems [1], GUI applications under Windows [2] and Apple MacOS [3]. Thanks to his simplicity and his power, it is often used to improve the programming of everything, because it can reach parts of code, in some cases, that other testing tools never reach, this concept of testing is still used nowadays and his creator is still doing research over this. Fuzzing is a method that inserts unexpected data into input. In this paper some approaches are discussed and also some applications and some tools are showed.

Fuzzing

Fuzzing a.k.a. Fuzz Testing is a technique developed by Barton P. Miller and his students at the University of Wisconsin, USA in 1989. It was used in several applications like Unix systems [1], GUI applications under Windows [2] and Apple MacOS [3]. Fuzzing is a powerful technique which can find errors in different parts of code, is commonly used by developers because of his concept. Fuzzing is well used in differents things differs on the implementation or the objective but the concept is the same.

What is Fuzzing?

Its call Fuzzing to all of types of techniques for Software testing capable of create and send data sequential or randomise to one or more parts of a Software, pointing to detect flaws or vulnerabilities in that Software. It is common used like a complement to another more common types of Software testing because Fuzzing give coverage to Data failures and part of the code is executed that was never tested before, thanks to the combination of randomness and heuristical attacks. In general the most of the Fuzzers try to find problems like: Buffer Overflow, Integer Overflow and Format String.
Its possible to identify some stages in Fuzzing, this could be:

- the first stage is getting the Data for the inputs, it can be, to the Fuzzing tool a data file with the possible inputs like in 1, or the data is generated by the Fuzzing tool and then sended like in 2 this means that can create a randomise input on the fly,
- then comes the part where the data is send to the Objective, there is here to options,

one could be that Software is remote or and the other is over LAN.

- Finally the output is analysed or in more general terms the behaviour of the Objective is analysed.

Always this kind of errors are human errors in the programming that's why Fuzzing can be used to find problems in every code written by Persons.

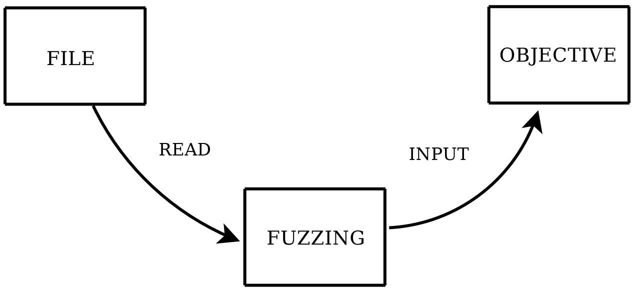

Figure 1: Read the input from a file

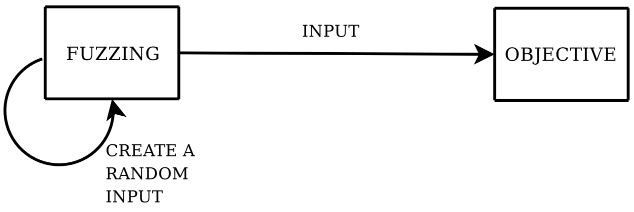

Figure 2: Generates the input randomly

State of Art

Call-Flow Aware API Fuzz Testing for Security of Windows Systems

Fuzzing try to put unexpected data in the input of a Objective that commonly is a Software, but in [4] Choi et. al. focus on API Fuzzing or API Fuzz Testing, that puts unexpected data in the function's parameters, and try to focus on API Applications in Windows XP SP2, and specifically to DLL in the System Folder *c:/Windows/System32*. There is a dependency of the functions in systems like Windows XP, that's why in this case there are errors that don't belong to the function itself but to the dependency, because of that in [4] make more intelligent the Fuzzing, for that this methodology, with the name *Call-Flow Aware*, before to call a function they resolve the dependencies first, to do that this algorithm is follow:

Algorithm of Call-Flow Aware API Fuzz Testing

1. Select API function for Fuzzing
2. Analyse automatically the dependency for API functions
3. Write C source codes that's calls target API functions with considering dependency of functions
4. Compile C source codes and make executables files
5. Execute executables files
6. Monitor Exceptions and errors

Is possible to identify 2 types of dependency between functions, this are:

- The return value of a function is a parameter for other function, e.g. two functions A and B, B depends of A if the return value of A (returnA), is a parameter of the function B, B(..., returnA,...) (see 3).
- A parameter of a function is related to the parameter of the other function, e.g. two

functions A and B, and there are a parameters paramA and paramB, B depends of A if they have related parameters, this means that A(...,paramA,...) and B(...,paramB,...), paramA and paramB have some relationship, this means that paramA in some way defines paramB (see 4).

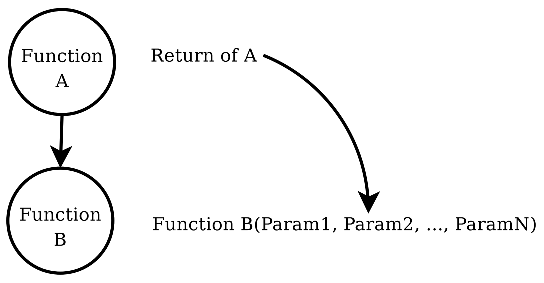

Figure 3: Return of A is a parameter of B

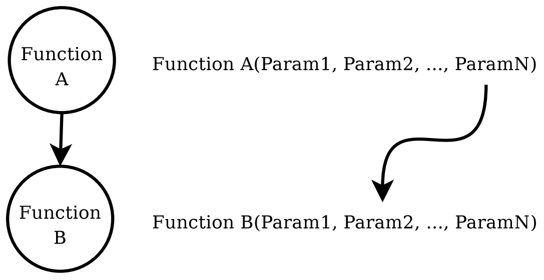
Function A
Function A(Param1, Param2, ..., ParamN)
Function B
Function B(Param1, Param2, ..., ParamN)

Figure 4: When to parameters are releated

This means that if its wanted to test for example a Function C, and this depends of B, and B depends of A, it should identify the type of dependency, then it can test correctly the Function C.

For all of this is it chooses to split this concept in three parts, the first one, is the one who trace all the functions called when a DLL is called (CFTR, Call-Flow Tracer) and creates a log file. Then the other part of the code this intelligent Fuzzing is the CFRO, Call-Flow Reorganizer, this takes the log file of the CFTR, and generates some rules. Finally the third part is the CFAFTR, Call-Flow Aware API Fuzz Tester, this is puts the invalid or unexpected data into the parameters of the functions using the some rules created for the CFTR, and monitors the behaviour of the software system.
With this Call-Flow Aware API Fuzzing it possible can review some code that is the intention of Fuzzing, those errors that were found in the first view of the DLL functions (functions with dependencies) were solved by the inclusion of this tool.

Tag-Aware Text File Fuzz Testing for Security of Software Systems

With Fuzzing it is possible to find the 20 or 25% of the security errors of Microsoft Applications before these were shipped[5]. Some of the Software Applications rejects the input data generated for the Fuzzing that makes it the inefficient. The effort to make the Fuzzing more intelligent, is done by building a structure of modules that first of all takes data from a real document, in this case it was spreadsheet from MS Excel exported to HTML with this is possible to create a HTML spreadsheet to import in this application, and with this information the Tag-Aware Fuzz Testing will learn about the file format, but this is only an example and very useful but is a way to probe the functionality of the methodology. The steps of this algorithm:

An algorithm for Tag-Aware Text file

Fuzz Testing

```
1. Generate a general text file
2. Make data of a text into one long string (OLS)
3. Search delimiters and information of attributes of tags in database based on a file extension
4. Extract tags in OLS using delimiters
5. Analyse— tags using information of attributes
6. For each $t_i \in$ tags
7.   For each $a_j \in$ attributes
8.     Inserts semi valid data in value of $a_j$
9. Monitors faults of a software system
10. Analyse a result of a fault
```

There are four parts on this Fuzzing, first the File Parser that take the tags of the files, then this information is used for the Fault Inserter, to create the invalid data or unexpected date, and finally the Fault Monitor, that observes the behaviour of the Software System.
In general the Fuzzers may have or not the awareness of the file format that why they sometimes can not fulfill successfully with the idea of reach code that other testing tools use, but existing tools that have some awareness of the file format but they aren't automatic.

Automated Whitebox Fuzzing

First of all, Blackbox Fuzzing uses the mutation of a well-formed input to put in the Software, and to do this is used grammars[6]. Whitebox Fuzzing instead of using grammars uses a symbolic execution and dynamically test generation, the intention with this is to trace how the application uses its inputs and with that information should be possible to know the path of the entire application, but this solution has some limitations.
Path explosion. If the application is only a few lines of code is very easy to know the different paths of this application, each increment of the size of the application means that the different can grow exponentially, this makes almost impossible to follow or to know all the different paths.
Imperfect symbolic execution. The symbolic execution becomes very difficult to follow if the application have complex statements, for example pointer manipulations, library functions and so, this means that having a good symbolic execution with reasonable cost is too difficult. For this limitations in [6], it is proposed:
Firstly, analyse the functions, his inputs and outputs and save those, this will be the functions summaries, but the problem with this is that the are not accurate, because is should be a complete path generation, but the time for generating this paht is the problem.
Secondly, it is obvious that the symbolic execution isn't possible, it is necessary to use a partial symbolic execution, with random values. It is possible that a vector input wouldn't follow the path predicted with this method, this means that a divergence has occurred. For that it is feasible to note this with checking the summaries and the actual behavior.
It is developed an algorithm called SAGE (for Scalable, Automated, Guided Execution). This algorithm is designed to:

- explore partially and systematically the states of a large application (thousand of symbolic variables) and very deep paths (millions of instructions)

- try to maximize the number of test generated from each symbolic execution, and tries to avoid redundancy in the search,
- use heuristics to maximize the code coverage to find bugs faster
- and be resilient to divergences, this means that it can recover from a divergence and continue.

This system is very promising, because in an early stage of this algorithm could find some errors on an application that was reviewed with Blackbox Fuzzing.

SAGE's Architecture
SAGE performs his generational search by four tasks, the first one is called *Tester*, this one checks for errors, if it is the case saves the test case but only continues if doesn't find any error. Then next task is called *Tracer*, this task as his name says, trace, with the same input as in the last task, the execution of the program, but in machine instruction level. The third task used is named *CoverageCollector*. This replays the same input as the last task but this time is checking which block of code are executed. And the last task is the *SymbolicExecutor* that executes once again the input and solving the symbolic constraints.
SAGE uses a machine based approach because of three reasons, the first one is for the large amount of different languages, if SAGE was a language specific tool, this means that would able to cover a little amount of applications, other thing is by using symbolic execution on compiled tools can help to find bugs in the compilation tools and post-processing tools and Finally by the unavailability of the source code of the applications.
About the results of applying SAGE, there a lot of things to say, they discovered that the divergences are very common and the bugs found were no in a high depth of the code and the heuristic wasn't so effective in coverage, that why in the next paper the authors try to improve this Whitebox Fuzzing adding the concept of grammars.

The Grammar-based Whitebox Fuzzing[7] instead of the old Whitebox Fuzzing uses tokens (the other one used only bytes for keeping the execution tracks) with this tokens should take out the invalid inputs, and with this may take out some coverage of the early depth of the program but doesn't happen. With the inclusion of the grammar for the symbolic constraints to generate valid inputs exploits the best of both Fuzzing, the blackbox and whitebox, and the coverage of the last one is incremented, and with this the results so much better. In [7] they compare the three Fuzzing, the Whitebox, the Blackbox, and the Grammar-based Whitebox but later after analyse the performance and the logic of this Grammar-based Fuzzing this takes as it said the best of both.

About Fuzzers

Fuzzers is the name of all the applications, tools, library or etc. that implements Fuzzing or used to create some Fuzzing tool.
A popular Fuzzer and Free hat the name *Peach*[1] and is a Fuzzer Framework written in Python. Peaches main goals include: short development time, code reuse, easy of use, and flexibility. *Peach* can fuzz just about anything from .NET, COM/ActiveX, SQL, shared

[1]`http://peachfuzzer.com`

libraries/DLLs, network applications, web, etc.
Other very famous tools is *SPIKE*[2], is an easy to use generic protocol API that helps reverse engineer new and unknown network protocols. It features several working examples. Includes a web server NTLM Authentication brute forcer and example code that parses web applications and DCE-RPC (MSRPC).
In [8] they used a library in Ruby, called *RFuzz*, and with that they create their own Fuzz tool to attack websites where is an user form as input. But the results aren't so conclusive to say that the tool that they create with this library is good enough, in other way this can says that is possible to find several errors in web pages with this characteristics, and than improve the programming.
For Web Applications there is a Fuzzer tool called *WebFuzzer*[3] and checks for remote vulnerabilities such as SQL injection, cross site scripting, remote code execution, file disclosure, directory traversal, PHP includes, shell escapes and insecure Perl open() calls.
Other in this same category is the *Cfuzzer*[4] is a simple C-source Fuzzer to test for HTTP chunked encoding issues in clients and servers.
Exist some other Fuzz for FTP protocol for example is called *FTPFuzz*[5] and is a simple GUI-based Fuzzer for testing FTPD server implementations. It allows the user to specify FTP commands and parameters to fuzz, and the pattern of test strings to use for each case. Remotely exploitable vulnerabilities in many popular FTP services have been discovered using this utility.
ISIC[6] is a suite of utilities to exercise the stability of an IP Stack and its component stacks (TCP, UDP, ICMP et. al.) It generates piles of pseudo random packets of the target protocol. The packets be given tendencies to conform to. Ie 50% of the packets generated can have IP Options. 25% of the packets can be IP fragments - but the percentages are arbitrary and most of the packet fields have a configurable tendency. The packets are then sent against the target machine to either penetrate its firewall rules or find bugs in the IP stack. ISIC also contains a utility generate raw ether frames to examine hardware implementations.
One is very interesting to mention is *ip6sic*[7], this one is a tool for stress testing an IPv6 stack implementation. It works in a way much similar to ISIC above. It was developed mainly on FreeBSD and is known to work on OpenBSD and Linux. Theoretically, it should work wherever libdnet works.
msn fuzzer[8], C source code for a simple MSN protocol fuzzer. Maybe it is used to discover vulnerabilities in MSN client software.
There is a lot of tools and frameworks avialable for Fuzzing.

Conclusion

Fuzzing is a tool often used and it is very powerful, but in the most of the cases or in simpler cases is a brute force testing tool. in the papers reviewed here is possible to know what is the intention of this authors in to create a more intelligent Fuzzing to improve the performance or to focus this testing in other direction, but to make this implies a lot of complexity, but is possible to find some improvements.

[2]`http://www.immunitysec.com/resources-freesoftware.shtml`
[3]`http://gunzip.altervista.org/`
[4]`http://www.open-labs.org/cfuzzer.c`
[5]`http://www.infigo.hr/files/ftpfuzz.zip`
[6]`http://www.packetfactory.net/Projects/ISIC/`
[7]`http://ip6sic.sourceforge.net/`
[8]`http://archives.devshed.com/forums/security-104/tiny-msn-fuzzer-passwd-demo-1253855.html`

Fuzzing is an interesting research area, there are a lot of tools for a lot of applications and protocols, this means that always the developers sometimes commit the same mistakes, that's why Fuzzing offers a possibility to improve the development of applications, in some cases the 20% or 25% of bugs are discovered with this technique. In cases of Microsoft applications is very important to search for bugs because always there are persons that wants to exploit this bugs.

References

[1] B. P. Miller, L. Fredriksen, B. So: An empirical Study of the Reliability of UNIX utilities, 1990, Comunications of the ACM Volumen 33 pp. 22

[2] J. E. Forrester, B. P. Miller: An empirical Study of the Robustness of Windows NT Applications Using Random Testing, 2000, 4th USENIX Windows Systems Symposium.

[3] B. P. Miller, G. Cooksey, F. Moore: An empirical Study of the Robustness of MacOS Applications Using Random Testing, 2006, First International Workshop on Random Testing.

[4] Y.Choi, H. Kim, H. Oh, D. Lee: Call-Flow Aware API Fuzz Testing for Security of Windows Systems, 2008, International Conference on Computational Science and Its Applications.

[5] Y.Choi, H. Kim, H. Oh, D. Lee: Tag-Aware Text File Fuzz Testing for Security of Software Systems, 2007, International Conference on Covergence Information Technology.

[6] P. Godefroid, M. Y. Levin, D. Molnar: Automated Whitebox Fuzzing, Technical Report MS-TR-2007-58, Microsoft, 2007.

[7] P. Godefroid, A. Kiezun, M. Y. Levin: Grammar-based Whitebox Fuzzing, 2008, ACM SIGPLAN Volumen 43 pp. 206-215.

[8] R. Hammersland: Fuzz Testing of Web Applications, 2008

LaTeX