Building a Fuzzing Framework: A Primer for Software Testers

Rahul Verma
McAfee Software India Pvt Ltd.,
Bangalore, Karnataka, India
rahul_verma@mcafee.com
rahul_verma@testingperspective.com
Author's Website: www.testingperspective.com

Abstract:
Fuzzing is a buzz word today. With its origin from the world of academic projects, it has found its way to the heart of companies like Microsoft, McAfee etc and many independent security researchers. It has become an essential part of the Security Development Life Cycle in many organizations and is known to find a high percentage of security issues as compared to other techniques.

This paper discusses basics of fuzzing (aka fuzz testing) and designing a fuzzing framework. It gives guidance for practical implementation of the same in different sorts of projects. The discussion will include the steps of fuzzing and required knowledge, how fuzzing frameworks are implemented, fuzzing decisions and the challenges involved.

Keywords:
Fuzzing, Fuzz Testing, Security Testing, Test Framework

1 Introduction

Wikipedia defines fuzzing as:

"Fuzz Testing or Fuzzing is a software testing technique that provides random data ("fuzz") to the inputs of a program. If the program fails (for example, by crashing, or by failing built-in code assertions), the defects can be noted."

As indicated by its definition, fuzzing is all about sending malformed data as input to an application to locate bugs. Such bugs typically result in crashes which after analysis can result in finding a vulnerability which makes the software exploitable in a certain way. A commonly discussed example of this sort is a buffer overflow vulnerability which can allow an attacker to inject shellcode in the application at run time and make it execute malicious code e.g. launching a remote shell.

Fuzzing is essentially an automated testing technique. As the number of test cases executed can quickly become very large, it is an art to carry out fuzzing with focus on areas which have the maximum possibility of locating vulnerabilities. It involves prioritization of tests based on analysis of the application, related protocol(s) and past vulnerabilities in similar applications. A common way of this is to make use of ***Fuzz Heuristics***, which refers to some known problem causing inputs of various sorts For example, long strings, long strings with format specifiers, integer boundaries, directory traversal strings etc.

Because of the aforesaid reasons, fuzzing as a testing technique is in evolving stage despite a large number of tools and frameworks developed at universities to start with and then by security researchers and vendors.

2 Boxing the Fuzzing Technique

Fuzzing has been considered to fit in the category of gray-box testing because of the nature of analysis and automation involved and can be executed as a black-box testing as well. The irony is that despite this fact, the term and the related implementation is mostly unknown to the software testers. It might be known to some testers working in companies like Microsoft, McAfee, and Cisco etc who have standardized upon a Security Development Lifecycle within the organization.

There is good chunk of fuzzing work which can be taken up by a software tester as against the general view of this being suitable only for security researchers. When tester locates a bug as a part of his usual job, he or she is rarely responsible for analyzing what piece of code is actually responsible for the bug. A tester usually logs the defect with test case details and his or her preliminary thoughts and analysis from outside the box. Fuzzing is no different, the only difference being the nature of the data that is submitted.

Fuzzing is usually thought of as a technique to find vulnerabilities, but it is also suitable for finding unexpected behavior on part of the application. A fuzz test might not always create a crash; if it makes the application ignore/consider a malformed input, this in itself is a bug and can be reported by the tester. This modified perspective about fuzzing brings it one more step closer to the job of a software tester.

Fuzzing makes a software tester think beyond BVA and ECP, makes him redefine his view of input to the application and extends the traditional approach to testing, by bringing in a lot of possible test areas.

3 Types of Fuzzing and Existing Tools

Knowledge about what already exists in the area of fuzzing helps one to understand practical implementations of different types of fuzzing. This helps in using or extending existing open source tools or coming up with altogether new tools and frameworks by analyzing the code and execution methodolgy of the existing ones.

Some of the common fuzzing types and the related free/open-source tools are listed below:

- **File Fuzzing:** FileFuzz[4], SpikeFile[4], NotSpikeFile[4]
- **Browser Fuzzing:** Mangleme[5]
- **Command Line Fuzzing:** iFuzz[2]
- **Environment Variables Fuzzing:** Sharefuzz[6]
- **Web Application Fuzzing:** WebScarab[7]
- **ActiveX Fuzzing:** AxMan[8], COMRaider[4]

In addition to the mentioned tools, there are also available, general-purpose frameworks which can be used to develop different fuzzing tools on top of them. The popular open source frameworks include Peach[1], Sulley[2] and SPIKE[3] fuzzing frameworks.

[1] http://peachfuzzer.com/
[2] http://www.fuzzing.org/fuzzing-software
[3] http://www.immunitysec.com/resources-freesoftware.shtml
[4] http://labs.idefense.com/software/fuzzing.php
[5] http://freshmeat.net/projects/mangleme/

[6] http://sharefuzz.sourceforge.net/
[7] http://www.owasp.org/index.php/Category:OWASP_WebScarab_Project
[8] http://www.metasploit.com/users/hdm/tools/axman/

The language used for development of fuzzers is more of a choice than a condition. The trend shows that the older tools were developed in C. As of current scenario, Python has become the language of choice of many security researchers and same is the case for fuzzing, though you can find a handful of tools in Java, C#, and Perl as well.

4 Step-by-Step Approach for Fuzzing

Fuzzing is easier understood if we split the process into steps. The fuzzing process can be remembered with **TIGEMA** mnemonic, wherein:

- T – Target(s)
- I – Input Vectors
- G – Generate
- E – Execute
- M – Monitor
- A – Analyze

Each of the above describes a distinct step in the process of fuzzing. One or more of them might work in conjunction or parallel to each other. Figure 1 gives a visual snapshot of these steps in conjunction with each other. Following sections discuss the steps in detail:

4.1 Identify Targets

The type of the application chosen is a major governing factor for the design of the fuzzing framework.

You might choose to fuzz the application you are currently testing or a third party application out of your interest. Testing a third party application will require you to have a local copy of the application in a test environment, as the basic purpose of fuzzing is to make the application crash. For a standalone application, this is the case anyway, but while testing a network service or web application, this point should be taken care of, without fail, to avoid any legal issues.

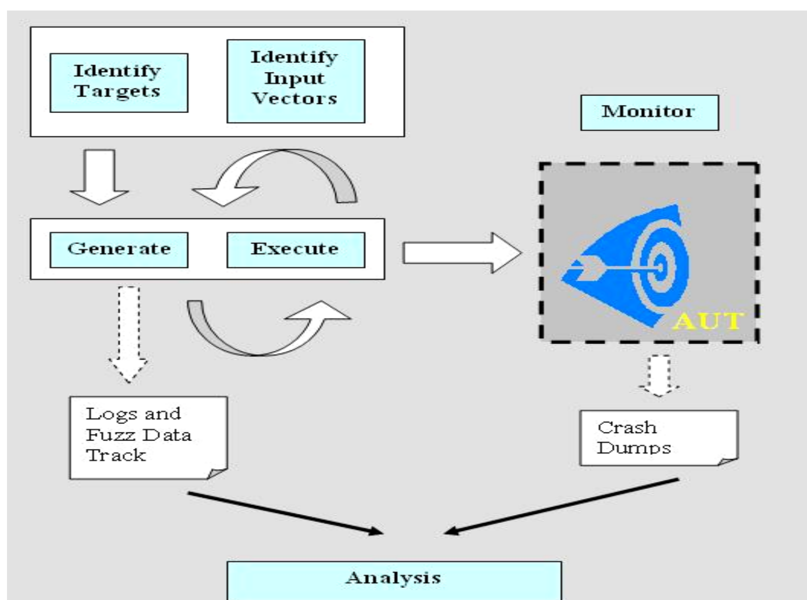

Figure 1: Steps in Fuzzing

4.2 Identify Input Vectors

For each target chosen, you should carefully analyze the inputs that it takes.

A tester's usual view of the input is something typed from the keyboard. The term input extends much beyond that. One should take into account all inputs given via a command line (if it has a command line interface), registry entries it reads from (on a Win32 platform), license files, data files, inputs from the GUI for applications (or a Web browser for web applications), cookies, client side controls (like ActiveX) and so on.

The nature and number of input vectors vary from application to application. The suggested way to do this is to do Threat Modeling for the application which is sometimes called Qualitative Security Analysis. This is further discussed in Section 7 – "Beyond the Basic Fuzzing Framework".

At this step the protocol used by the application and the file formats are analyzed. The level of analysis depends on the kind of fuzzer you are trying to build. Section 3 - Framework Decisions talks more about these considerations.

4.3 Generate Fuzz Data

This is the step where actual fuzzer development comes into picture. Based on the inputs chosen, you make decisions for the kind of fuzzing you want to employ. This governs the quality and quantity of fuzzed data you will receive for the inputs you have identified. . Section 3 - Framework Decisions talks more about these considerations.

4.4 Execute

At this step you send (publish) the fuzzed data (for an input vector) to the target application. This might be a post generation process or it might run along with the generation process.

In the former, you first generate all the fuzzed data and write to an output file and later send this data one by one to the application. You might require a lot of disk space in this case depending on the kind of fuzzing. E.g. if you are fuzzing file formats, if the size of the file is large, you might end up consuming a lot of disk space (or at the worst running out of disk space). In some cases, this option might not be feasible at all.

In the latter case, you generate fuzz data and send it to the application. In case the application crashes, a copy of the data is retained and next fuzz iteration gets executed; else the data is ignored (or deleted if on disk) and fuzzing process is continued. This way, only that fuzz data which is problematic is retained on the disk (in the form of files/database entries etc.). This again has limitations in case of state-based testing where a series of inputs is causing a crash, which is out of scope of the present discussion.

4.5 Monitor

This is done while you are sending the data to the application. This typically involves a debugger being attached to the application right from the beginning of the test. It might also involve monitoring the resource utilization on the box. If there is a crash, the fuzzer should be able to know about it. The debugger takes a dump of the application in case of a crash for later analysis. The fuzzer then launches the application again, attaches the debugger and proceeds to the next fuzzing step. Section 7 – 'Beyond Basic Fuzzing Framework" - talks about Crash Dump Analysis.

In some subtle case, where a deadlock occurs, no crash dump is taken. So, the fuzzer should have a component which puts a cut off limit on the running time of the application (called time threshold), monitor the related process and kill it if required. If the application is expected to finish execution before the time threshold and exit, all such instances where the application did not finish execution could be caused due to some infinite loop or similar condition and should be analyzed. Time threshold also helps in killing an application and proceeding with the next test case as a part of normal fuzzing process (e.g. in case of file fuzzing).

4.6 Analyze

The crash dump and the fuzz data that caused it are taken for analysis at this stage. This is typically taken up by a security researcher and/or development team with knowledge of vulnerability analysis.

The software tester's job at this stage is providing the required data to the mentioned team. Based on interest, a tester can learn basic crash dump analysis and be of further help.

5 Framework Decisions

There are a plenty of open source tools available out there. You might choose to use one of them directly or extend it to suit your requirements. As this paper focuses on building a basic in-house framework, we will not go into details of the available frameworks. But the knowledge shared herein will be helpful in both ways, whether you want to build your own framework or to understand and utilize an existing one.

When you choose to build your own framework, there are a lot of questions which must be answered before starting the development work for it:

5.1 Do you want to fuzz just a particular product or a set of products across the organization or some third party products out of your interest?

This should tell you whether to go for a generic framework or a product-specific fuzzing tool. Usually a framework involves a lot of design decisions and is relatively complex as compared to a product-specific fuzzer, but has the advantage of reusability.

5.2 What type of application are you targeting?

The nature of the application (desktop/client-server), platforms supported (Windows / UNIX-Linux), its type (command-line/GUI-based/browser plug-in/API) etc. have a lot of impact on the way fuzzing is done.

For fuzzing a server listening over the network, you will have to simulate the protocol that its corresponding client uses to communicate with it. You might need some sniffer component, for which you need to use a third party sniffer or build your own. In the former case there are integration issues that need to be addressed and for the latter, you should have resources with optimum knowledge. It will govern the protocol support that you have to build in your framework. If it is a specific application, you will have to build application-interaction functions in your framework. Similar considerations exist for command line fuzzing, file format fuzzing, web browser fuzzing etc.

Also, the type of OS platform has a large impact on the way the tool is designed because of the fuzzer needs to understand how the OS handles process, what the available debugging options are, how resources can be monitored etc.

5.3 At what layer, do you plan to employ fuzzing?

It governs the parsers you will hit with fuzzed data. E.g. if you are fuzzing a web application, you may not be able to test the application properly if you are carrying out fuzzing by automating the internet browser. The browser will have its own checks on the data. In addition, JavaScript and other client-side checks might prevent you from sending data of various sorts. In such a case, you have to fuzz at the HTTP level.

5.4 Do you plan to produce fuzz data by generation or mutation?

At a broad level, a fuzzer can produce fuzz data in two ways – generation and mutation. In generation, the complete protocol is generated from scratch based on the knowledge of the protocol built into the fuzzer. This requires lot of ground work to be done by reading relevant manuals and analysis. In case no such published data is available, you will have to resort to reverse engineering skills, which most of the times is quite a complex task. The advantage is that you get complete control over the protocol and can get good code coverage.

Mutation is about capturing good data and then fuzzing various sections of data. For this you might employ, for example, a network sniffer to capture “good” network packets (for network protocol fuzzing), use a base good file for mutating (for file fuzzing) etc. The advantage is that you can get started with fuzzing efforts quickly, but one must take care of internal dependencies of the fields and optimum code coverage.

5.5 Do you want the fuzzer to abide by protocol's internal checks?

In many protocols there are fields that are dependent on other fields in turn, e.g. they might include length, checksums etc. If you choose to abide by these conditions, the fuzzing process gets a little trickier than otherwise. A suggested way is to build these checks into the fuzzer you build and carry out tests in both ways – breaking the dependencies and abiding by them. This helps to unearth any false assumptions and also making sure that correct parsers are hit (of course by increasing the number of test cases executed significantly)

5.6 Do you want to build a blind fuzzer or a protocol-aware fuzzer?

A blind fuzzer has no knowledge of the underlying protocol. It is assigned the task of blindly corrupting or generating a data packet and sending to the application. This is very easy to build but results in wastage of CPU cycles and time in generating and testing data that is out rightly rejected, sometimes, much before it reaches the target. The protocol aware fuzzer is complex to build but is more reliable and result-oriented.

Blind fuzzers are usually tied to the mutation approach and protocol aware fuzzers are tied to generation approach (or to mutation approach while abiding by the dependencies of the fields)

6 Framework Components

Figure 2 gives a visual snapshot of the various components of a fuzzing frameworks and how they work together. Following sections discuss the components in detail:

6.1 Protocol Builders

They deal with defining the protocol one is trying to fuzz. A protocol in this context can be a network protocol or a file format. Protocol builders define the protocol field by field taking into account the dependencies, if any, of one field on other field(s). They might also associate at choice, a set of fuzzers and corresponding Morphers to the field. The fuzzers and Morphers also might be chosen by default by the fuzzing framework. Ideally, such a control should be given in the hands of users – use default fuzzers associated with a field type by the framework or override the same with custom settings.

6.2 Fuzzers

The purpose of fuzzers is to generate data or mutate the provided data.

The data generating fuzzers generate data as per pre-defined algorithms. What they generate how many combinations they generate, whether it is random or growing data – all these form the basis of different set of fuzzers.

The second category i.e. the mutating fuzzers mutate the provided initial data. One of the practical applications of them is file fuzzing. In file fuzzing, a file format is input to the fuzzing engine which is then corrupted field by field and input to the AUT (Application under Test).

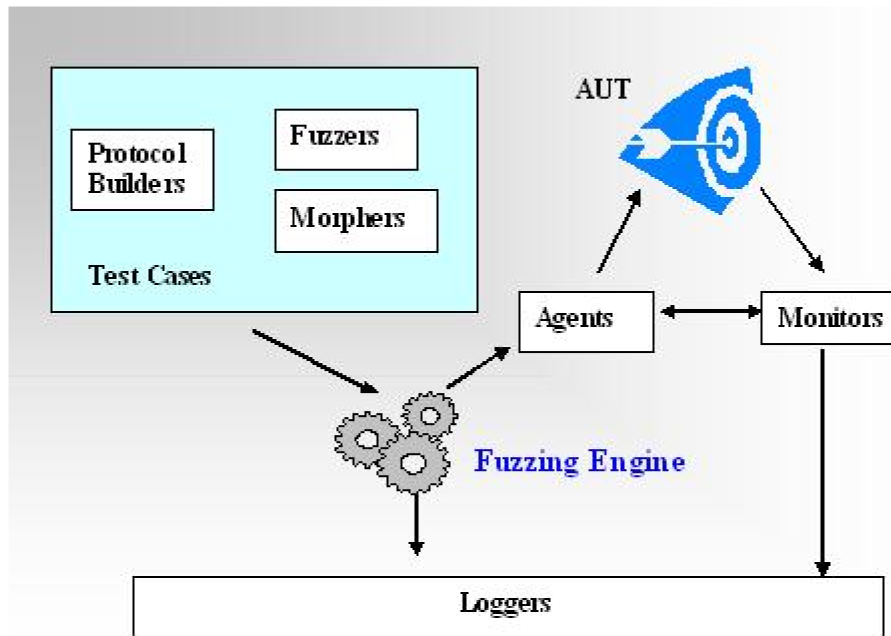

Figure 2: Fuzzing Framework Components

6.3 Morphers

Most of the times the protocols require some of their fields that are not in clear text format, or require encoding or compressing of some sort.

Morphers modify data by encoding/encrypting/binary packing/type conversion. The output of the morphers is the data that is actually used while sending the data to the AUT.

6.4 Test Cases

Test Cases define the combination of a protocol builder and the associated set of fuzzers and morphers. Typically, many such combinations exist because of the 'What to fuzz?' and 'How to fuzz decisions?'.

6.5 Agents

Agents help the framework to deal with applications, OS and network. Typically, a framework will have a file handling agent, application specific agents, and protocol publishing agents. Agents interact with the application to carry out the fuzzing.

6.6 Monitors

Monitors monitor the application for crashes during the fuzz test. They can also monitor the processes, the time a process takes for completion and resource utilization. Monitors can be a part of the agents or separate full fledged debuggers.

6.7 Loggers

Loggers are used to log the information in a way that can be taken at a later stage of further analysis. Loggers take input from the Fuzzing engine to log the fuzzing data. They also take input from Monitors regarding any crashes.

In case one is using a third party debugger, integrating it with the Logger component becomes a problem unless it exposes an API to suit this. If you implement debugging based on an API (third party or base OS process API), this step is achievable. In actual terms, this is the suggested approach to have one-to-one association between input and output.

7 Beyond the Basic Framework

For carrying out fuzzing in an effective way a lot more knowledge is required about protocols, application internals, OS internals etc. These tasks might be taken care of by people who actually develop the fuzzing framework (if you are using an existing one) or by security researchers in the analysis phase, but it always helps to build knowledge in related areas. Following are specifically some areas that I would suggest to work upon for any software tester taking up fuzzing or any other form of security testing seriously.

7.1 Threat Modeling

Threat Modeling is carried out to locate all the possible interfaces of the application for taking inputs of various sorts. For each of these interfaces, all input vectors, corresponding threats and roles are associated. At each step the risk is given weight in terms of probability of occurrence and degree of damage.

The above exercise is helpful for fuzzing because it tells you about the interfaces that you can fuzz, the kinds of malicious inputs. It also helps you in prioritizing tests based on the risk factors.

7.2 Code Coverage

Code coverage gives you a means to measure how effective your fuzzing was in exercising the code. It might be done by choosing suitable compiler-induced code instrumentation at compile time, if you are yourself compiling the code of the application under test. There are other (rather complex) ways for binary code coverage with the help of binary debuggers.

7.3 Crash Dump Analysis

In the event that you are actually able to crash the application with fuzzing, a crash dump is to be created and submitted for analysis to the developer or preferably a security researcher. Basic knowledge of crash dump analysis helps a tester to report the issue better and at times conducts a series of related tests to dig further issues in the application.

7.4 Vulnerability Analysis

As an extension to the previous step, when you have found what caused the crash, you can take it one step further by analyzing whether the bug is actually exploitable. In such a case, the bug becomes vulnerability. Vulnerability is a bug in the application which can be exploited by a malicious user to make the application perform operations not intended E.g. opening up a remote shell, escalation of privileges, bypassing security checks etc. Such problems are more serious than a simple crash.

References

[1] Sutton Michael, Greene Adam, Amini Pedram: 2007. *Fuzzing: Brute Force Vulnerability Assessment.* Addison-Wesley Professional.

[2] Eddington, Michael: 2008: Peach 2 Tutorial.
http://peachfuzzer.com/docs/Peach%202%20Tutorial.htm

[3] Wikipedia: Fuzz Testing.
http://en.wikipedia.org/wiki/Fuzz_testing

[4] OWASP: Fuzzing.
http://www.owasp.org/index.php/Fuzzing

[5] Fuzzing.org: Fuzzing Software.
http://www.fuzzing.org/fuzzing-software

Biography

Rahul Verma

Rahul Verma is currently working with Mcafee India, in the AVERT team. With more than five years of industrial experience, he is well equipped with software testing concepts and various open-source/commercial tools related to performance, security and functional testing for web applications, web services and thick client applications. He has presented several technical and non-technical topics so far. More about him can be found at his website www.testingperspective.com.