Fuzzing—or how to help computers cope with the unexpected

Testing the security of software before implementation can be a hit-and-miss affair, and is unlikely to discover every weakness. **Toby Clarke** and **Jason Crampton** explain how the technique of fuzzing can help in all stages of testing.

Royal Holloway University of London
SearchSecurity.co.UK

THE PRESENCE OF vulnerabilities in software applications represents a significant risk to the security of individuals, businesses and nation states. As increasing reliance is placed upon data and the systems that process and store it, the risk associated with vulnerabilities in software applications increases.

In this article we will examine fuzzing: a specific type of software security testing which can be used to discover vulnerabilities in software applications.

We will begin by looking at software vulnerabilities and security testing in general, before focusing on fuzzing. After describing a basic model of a fuzzer, we will identify and discuss four fundamental problems that practical fuzzers have to address. We will then explore three different approaches to a key aspect of fuzzer design: fuzz test data generation. Finally, we will examine what fuzzing can, and cannot, offer us.

Humans are rather good at coping with the unexpected, but computers, by default, are not. In the early days of computing, programs were used exclusively by thoughtful, scientific people who were very methodical and precise about entering data.

As computers became more common, a wider range of people began to interact with computer programs and data entry errors often led to program failures. The phrase "syntax error" drove many users to distraction.

Developers tried to defend themselves with the rallying cry "Garbage in, garbage out", and the age of the Graphical User Interface was ushered in to address the failings of the Command Line Interface (or as a means to contain the damage users were capable of, depending on your viewpoint).

However, computers and computer programs were going to have to get a lot better at dealing with unexpected data, because the wider population of humans were not only error prone, but sometimes actively malicious. Once computers became networked, software had to learn to defend itself.

SOFTWARE VULNERABILITIES

Software vulnerabilities occur when an application permits the confidentiality, integrity or avail-

ability of data it processes or stores to be affected in an unauthorized manner. In other words, a software vulnerability is a security compromise waiting to happen: an attacker can exploit the vulnerability in order to read or modify the data the software processes, or even take control of the computer system on which the application resides.

Vulnerabilities can stem from failings at all stages in the life cycle of an application. This includes the software design and implementation (programming) phases of development, the deployment and operational stages, right through to the retirement and removal of an application.

Programming defects ('bugs') are of particular concern, since they are so common[1] that it is usually not economically viable to detect and correct every bug during the development of an application.

Instead, bugs are typically rated in terms of the risk they represent, and resources are allocated to addressing the most critical bugs until an agreed 'bug bar' of acceptable bugs is reached.

While it may be acceptable to release a commercial software application with hundreds or even thousands of known bugs, bugs that have potential to be exploitable (i.e. vulnerabilities) must be detected and corrected as far as possible. As well as protecting users and their data, this makes economic sense, since the cost of correcting vulnerabilities post-release is far greater than doing so pre-release [5].

SOFTWARE SECURITY TESTING

The objective of software security testing is to identify the presence of vulnerabilities, so that the defects that cause them can be addressed. Security testing methods can be grouped into two key areas: static and dynamic testing.

- **Static testing** usually involves analysis of the high-level application source code, but can also be performed using low-level assembly 'dead listings' if the source is not available.

- **Dynamic testing** involves testing for vulnerabilities by analysing the application as it executes; this means that access to the source code is not a requirement. Examples of dynamic software security testing include *API fault injection*, where error messages are passed to the application at the API-level, or *run-time permissions auditing*, where tools such as Sysinternals *Accesschk*, *WinObj*, or *Process Explorer*[2] are used to test the application as it executes for inappropriately

weak access permissions—an example of this is provided by Cerrudo [3].

Fuzzing is a form of dynamic software testing that works by submitting malformed input(s) to the target application to see if it causes an error. If a software application crashes as a result of receiving malformed input, the fuzzer has discovered a software defect.

Fuzzing can be thought of as a form of software security performance testing. There are similarities between stress testing—*"Can this application handle 1000 simultaneous users?"*—and fuzz testing—*"Can this application handle a URL with 256 characters?"*. Indeed, the first ever fuzzer (appropriately called fuzz), was developed to test the robustness of UNIX applications [9]. However, when a fuzzer is employed for security testing, the objective is usually not to determine whether a functional requirement is met, but to trigger and detect software defects so that they may be addressed.

The term *fuzzer* may be used to describe a wide range of tools, scripts, applications, and testing frameworks. However, most fuzzers share the following features:

- Data generation (creating data to be passed to the target application);
- Data transmission (getting the data to the target);
- Target monitoring and logging (observing and recording the reaction of the target); and
- Automation (reducing, as much as possible, the amount of direct user-interaction required to carry out the testing regime).

One of the key aspects that differentiate fuzzers is test data generation. But before we examine this topic, it would be useful to understand some of the problems that have shaped fuzzer development.

FOUR FUNDAMENTAL FUZZING PROBLEMS

At least four fundamental fuzzing problems must be addressed to produce an effective fuzzer. The degree to which a fuzzer addresses these problems will significantly influence its success in discovering vulnerabilities.

1. Input Validation. Almost all commercial software applications incorporate some form of input validation. In order to ensure that test data is processed by the application, a fuzzer should

generate malformed data that can also satisfy validation checks if possible. This does not necessarily require analysis of the target application source code or design documents: validation checks may be documented in a network protocol or file format specification.

2. Optimising Program State Coverage. A running program can be thought of as a machine which moves through a number of states. Ideally, a fuzzer would ensure that every possible state is 'visited'' during testing, by programmatically generating and submitting every possible permutation of input (termed the input space) to the application under test. Unfortunately, it is infeasible to enumerate the input space of all but the most simple of programs. The best fuzzers attempt to optimise program state coverage, maximizing the number of potential security vulnerabilities they discover, while reducing test duration.

3. Detecting Errors. It is not enough to trigger failures: a fuzzer needs to be capable of detecting failures, otherwise they cannot be reported to the analyst.

4. Target Recovery. Once the target crashes, testing must stop until the target has been returned to its normal running state. It would be very useful to automatically return the target to its normal running state as this would allow unattended testing to be performed.

Solving the input validation problem involves creating input that is malformed (in order to cause a crash) while also satisfying any integrity checks that the application may employ to validate input such as:

- content length checks as found in many network protocols such as Hyper Text Transfer Protocol.

- checks for the presence of static values at specific locations such as Java's use of the CAFEBABE 'magic number'.

- self-referential checks such as cyclic redundancy checks (CRCs) found in the Portable Network Graphic (PNG) image file format.

Most applications employ checks to detect data corruption, as opposed to active data tampering. Such checks can usually be circumvented by making the fuzzer 'aware' of them. For exam-

ple, where static values are used, the fuzzer is informed of this (either through a GUI or some form of configuration script) and can then generate test data that includes the required value at the required location. Where length checks are used, the fuzzer would have to re-calculate these to ensure they correlate with malformed values.

Solving the program state coverage problem by enumerating all possible program states would require the generation of every possible combination of input, which is simply infeasible for modern software applications.

Instead, practical fuzzers focus upon triggering errors and thus discovering fault states. By accepting that we cannot enumerate the entire state space of an application, we may construct practical fuzzers that can complete testing within reasonable timescales, but we must also accept that there can be no assurance that they will detect all present bugs. In practice then, fuzzing is like fishing for bugs; if you find none, you have learned nothing. You cannot prove the absence of bugs via fuzzing.

In fact, all software testing is subject to this limitation. The International Software Testing Qualifications Board states that "Testing reduces the probability of undiscovered defects remaining in the software but, even if no defects are found, it is not a proof of correctness." [8]

Most fuzzing is limited to input/output analysis, and can be classed as 'black box' testing, since the internal workings of the target software are "invisible" to the tester. Extending fuzz testing to include analysis of the internals of the application (termed white-box fuzzing) provides opportunities to address some of the issues relating to black-box fuzzing.

Code coverage is one example of a white-box fuzzing technique that offers feedback about fuzzer performance. Code coverage involves measuring the amount of code that is executed during a complete fuzzing session, and comparing this with the total code of the application.

If a fuzzing session results in 10% of the total code base being executed, it is obvious that the fuzz test has not been very effective (or a large proportion of the code is not reachable from input).

If a fuzzing session results in 100% of the total code base being executed, we can say we have successfully traversed all possible code branches, but we can say nothing about what data was passed to the application at each code branch. Hence, although it can provide some useful feedback, code coverage does not necessarily provide

an accurate metric of fuzzer performance.

Solving error detection is achieved using an oracle, a generic term for a software component that monitors the target and reports a failure or exception. An oracle may take the form of a simple liveness check, which merely "pings" the target to ensure it is responsive, or it may comprise a debugger running on the target that can monitor for, and intercept, exceptions, and collect detailed logging information.

Solving target recovery may be as simple as automatically restarting the target application before each individual test case is passed to it. However, this may be time consuming when multiplied over a large number of test cases. An elegant approach is to integrate an oracle into the fuzzer and extend its functionality to allow it to restart the target if an exception occurs. This functionality is present in the Sulley fuzzing framework.[3]

Having described some of the requirements for effective fuzzing, we will describe three approaches to fuzzer test data generation and will explore how these approaches address the input validation and program state coverage problems. Although we will not cover exception monitoring or target recovery any further, these subjects are discussed in the technical report on which this article is based [4, Chapter 7].

FUZZER TEST DATA GENERATION

The objective of the test data generation aspect of a fuzzer is to induce failures in the target application. Almost all fuzzers create a large number of individual 'input buffers' or test cases, each one of which will be passed as input to the application, one at a time.

A number of different data generation approaches have been adopted, which can be grouped into one of the following classes: random, (blind) data mutation, and protocol-based analysis data generation.

Random data generation involves using a pseudo-random generator to produce random test cases, minimizing the effort and time required in initiating testing. However, random generation is the least effective of the three classes in terms of triggering bugs.

Using human language as an analogy, if I were to approach you uttering random vowels and consonants at you, you would probably ignore me. In order to engage people verbally, I (usually!) shape

sounds into words that they recognise and arrange those words into a sequence that gives them meaning; I obey lexical rules (use the right words) and grammatical rules (put the words in the right order).

Modern software applications often apply lexical and grammatical rules to determine whether input will be processed or rejected. Ideally, fuzzer test data should conform to lexical and grammatical rules to a degree sufficient to be accepted into the application, but should also violate either the same rules or cause some violation of the program logic, in both cases triggering a failure.

"Random generation fuzzing is arguably the least effective method of test data generation, since a large ratio of test cases will be rejected by the target application."

Random data generation is highly inefficient since it cannot account for lexical or grammatical rules applied to input, nor can it apply any intelligence in terms of triggering failures. That does not mean that random generation fuzzing does not work: many bugs have been discovered this way. However, this is arguably the least effective method of test data generation, since a large ratio of test cases will likely be rejected by the target application.

Data mutation brings together two important techniques: data capture and selective mutation. Valid input is captured and used as a means to quickly and easily define well-formed input. The captured data or mutation template is then selectively mutated so the test data produced contains much of the structure present in valid input.

For example, returning to our human language analogy, we might capture the valid introductory sentence *"Can you talk?"* and mutate it into *"Can you walk?"*.

In this way, we start out with a well-formed input that will likely be accepted by the target, and we can then selectively mutate it, creating boundary cases where input validation routines struggle to differentiate between valid and invalid input.

For example, if we are fuzz testing a web server, we might start with a valid HTTP GET request such as:

```
GET /index.html HTTP/1.1
```

Malformed requests could then be generated at the text level by simply replacing each character of the input string with, say, the '_' underscore character:

```
_ET /index.html HTTP/1.1
G_T /index.html HTTP/1.1
GE_ /index.html HTTP/1.1
GET_/index.html HTTP/1.1
```

This is a form of 'blind' fuzzing, since no knowledge of the protocol is required. By replaying and programatically altering an input buffer, we can create near-valid data, which has a number of benefits. Firstly, the efficiency is improved as the number of test cases which are rejected due to the program's inability to parse the input is greatly reduced compared to random data creation. Secondly, we can target specific aspects of input handling because we isolate specific aspects of input and mutate them.

In general, software application input invariably consists of a number of distinct data elements which are often processed by different parts of the application each using distinct techniques to parse the input.

When a web server processes an HTTP request such as the one above, it will probably separate it into three elements: 'GET' (the method), 'index.html' (the resource) and 'HTTP/1.1' (the protocol), each processed by a different routine. By selectively mutating a valid request we can produce test data with the potential to target specific processing routines. However, this is achieved without any awareness of the protocol.

Blind data mutation often fails to achieve good code coverage for a number of reasons. One flaw is that it is limited to mutating the captured data. If we capture and mutate an HTTP GET request in order to fuzz a web server, what about other requests such as POST, TRACE, HEAD and so on?

One solution is to capture multiple data samples and create multiple mutation templates, but while each new template has the potential to increase code coverage, it may increase test duration significantly.

Data mutation is a significant improvement over random fuzzing, since it can (albeit 'blindly') account for lexical and grammatical rules.

However, this 'blind' approach cannot account for checks over input data such as hashes or CRCs.

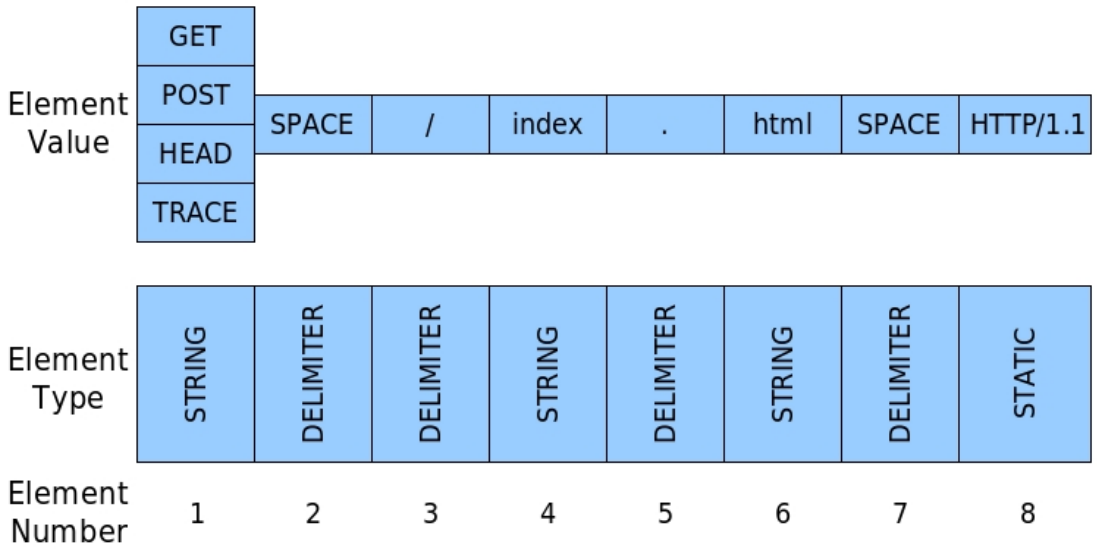
Protocol analysis-based data generation is based on the fact that application input data often conforms to a protocol of some sort. A 'protocol aware' fuzzer can be created by defining a model of the protocol so that the fuzzer can group and sequence data elements into valid input messages.

FIGURE 1 provides a graphical representation of a very basic protocol definition.

Starting with element 1, we see that four different values have been defined: GET, POST, HEAD and TRACE. This would allow the fuzzer to gener-

FIGURE 1

A GRAPHICAL REPRESENTATION OF A BASIC PROTOCOL DEFINITION



ate GET, POST, HEAD and TRACE requests, extending code coverage by satisfying lexical and grammatical rules.

Element 1 has been defined as being a *string* type. The fuzzer can use this information to replace the element with a range of mutation elements that are specifically aimed at triggering bugs in string handling routines.

> *"This block-based approach can be extended to include elements that describe aspects of other elements, allowing valid CRCs or hashes to be calculated for input buffers."*

Element 2 has been defined as a *delimiter* type. In general, fuzzers do not fuzz all elements simultaneously but selectively fuzz each element individually while replaying valid values for the rest of the elements. In this way a fuzzed element is followed by the correct delimiter, increasing the likelihood that it will be processed. The fuzzer can also replace this element with mutation elements that are known to cause problems with delimiters.

Element 8 has been defined as a static type. This element will not be fuzzed, reducing the number of test iterations.

Only a few element types are present in our example, but many more could be used to define data types such as integers, signed integers, characters, bits, etc.

This *block-based* approach can be extended to include elements that describe aspects of other elements, allowing valid CRCs or hashes to be calculated for input buffers. Content length descriptors and element size descriptors are featured in many protocols, and can also be constructed and deployed within a block-based approach.

SPIKE[4] is an excellent example of a block-based fuzzer framework (its creator, Dave Aitel, has also produced numerous valuable articles and presentations on fuzzing including [1] and [2]). My favourite fuzzer development framework *Sulley*[5] also applies a block-based approach to protocol definition.

A fuzzer that has a protocol definition can identify individual data elements and their types, and apply intelligence to triggering faults. This is often

achieved by deploying a database of distinct classes of known-bad data buffers (which could be termed mutation elements) and enabling the fuzzer to target individual element types with type-specific attacks.[6]

Protocol-based fuzzing is able to bring together a protocol definition and a database of known-bad buffers in order to create test data that is likely to result in a high degree of code coverage and have a high probability of triggering faults. Block-based fuzzing frameworks can be used to construct protocol-based fuzzers.

WHAT FUZZING CAN DO FOR YOU

Now that we have described what a fuzzer is and how it works, let's take a brief look at what fuzzing has to offer us:

NO REQUIREMENT FOR SOURCE CODE. What if you wanted to assess and compare the security stance of three different closed-source software products? You could ask for evidence of software development quality assurance, or examine the history of vulnerabilities in each vendor's products (although it could be argued that the more mistakes a vendor makes, the more the vendor learns about secure software development).

Fuzzing, like all forms of run-time testing, offers a means to discover flaws in software applications without access to the source code.

VIOLATION OF ASSUMPTIONS/SCOPE LIMITATION. What if, as a developer, an external component you are relying upon has a defect? What if your application exposes a vulnerable aspect of a third party driver or the operating system itself? What if, as a tester, your scope is limited?

One benefit of not having access to information about the inner workings of an application is that the test cannot be influenced by this information. There is always a risk that the scope of testing will be restricted based on misplaced trust; an example of this would be reviewing the source code of an application, but failing to review the source code of third party drivers or even the host Operating System itself.

Such scope limitation is usually impossible to achieve when fuzz testing. In fact, the behaviour of an application undergoing fuzz testing is highly unpredictable; this is why fuzzing should never be performed in a live, production environment.

The unpredictability of fuzzing, and the fact that it tests the application and the environment it executes in, prevents misplaced assumptions and scope restriction; fuzzing is capable of dis-

covering bugs in an application and any code it interacts with.

FUZZING IDENTIFIES REACHABLE BUGS ONLY AND PROVIDES PROOF OF CONCEPT CODE. All security testing methods (static or dynamic) are subject to false positives; they discover bugs that appear to be a security vulnerability, but are in fact not exploitable.

One class of false positives are potential vulnerabilities that can't be triggered by input, and hence are not exploitable. An example of an such an 'unreachable' vulnerably would be a call to a vulnerable function that is present in the program code, but can never be executed.

Because fuzzing works by sending malformed input, it cannot discover unreachable bugs, saving the analyst the effort of assessing each potential vulnerability for reachability. Fuzzing is perhaps unique among security testing methods in this aspect of its behaviour.

Additionally, since fuzzing reports the specific input buffer that triggered a given bug (usually by identifying the test case that caused the crash), fuzzers provide demonstrable, repeatable evidence of a defect—a crude form of proof of concept code.

Here's a look at what Fuzzing cannot offer:

FUZZING PROVIDES NO ASSURANCE. No testing technique can provide concrete assurance. However, fuzzing may be more likely to miss bugs than other testing techniques. A fuzzer may fail to trigger the relevant program state, it may fail to submit data sufficient to trigger a bug, and it may fail to detect a triggered bug.

Fuzzing can be used to discover bugs, not to provide assurance of their absence.

FUZZING IDENTIFIES BUGS, NOT SECURITY VULNERABILITIES. All dynamic testing, including fuzzing, aims to identify bugs. However, not all bugs are vulnerabilities. Skilled analysis of bugs discovered using fuzz testing is usually required in order to determine whether a bug is exploitable, and rate the associated risk. An example approach to ranking bugs discovered using fuzzing is provided by Howard and Lipner [7].

SUMMARY

We have seen that fuzz testing provides a method for discovering vulnerabilities by triggering software defects using malformed input(s) as the application executes. We have described four fundamental problems that a fuzzer should address, and we have explored three different

approaches to generating test data. Finally, we have suggested some of the benefits and drawbacks of fuzzing.

Regarding the comparison of fuzzing with source code analysis: it has been argued that source code review provides more assurance [11], and also that fuzzing finds bugs that static testing cannot [10]. I would argue that the two are complementary; that both offer unique methods to discover vulnerabilities, and that neither should be neglected.

The author would like to think that this article may encourage more IT professionals to consider how they might take advantage of fuzz testing, and to do so with a clear idea of its limitations as well as its benefits. I welcome any feedback, and invite readers to comment on any aspect of this paper. ■

ABOUT THE AUTHORS

Toby Clarke *is a security consultant providing penetration testing services within KPMG's IT Advisory Practice*

Dr. Jason Crampton *is a reader in information security at Royal Holloway, University of London, with research interests in access control. He teaches the course in Computer Security on the M.Sc. in Information Security at Royal Holloway.*

SOURCES:

[1]One estimate suggests that even when software development is subject to strict Quality Assurance controls, 5 bugs may be present per 1000 lines of code [6].

[2]http://technet.microsoft.com/en-us/sysinternals/default.aspx

[3]http://www.fuzzing.org/

[4]http://www.immunityinc.com/resources-freesoftware.shtml

[5]http://www.fuzzing.org/

[6] A detailed examination of a selection of the attack buffers employed by the SPIKE fuzzing framework (created by Dave Aitel) is included in Appendix B3 of [4].

REFERENCES:

[1] Dave Aitel. The advantages of block-based protocol analysis for security testing, 2002.

[2] Dave Aitel. MSRPC fuzzing with SPIKE 2006. On-line article, August 2006.

[3] Cesar Cerrudo. Practical 10 minute security audit Oracle case. On-line article. Argeniss Information Security, Presentation at Black Hat DC 2007.

[4] Toby Clarke. Fuzzing for software vulnerability discovery. Technical Report RHUL-MA-2009-4, Information Security Group, Royal Holloway, University of London, 2008.

[5] Erwin Erkinger. Software reliability in the aerospace industry—how safe and reliable software can be achieved. 23rd Chaos Communication Congress presentation, 2006.

[6] Greg Hoglund and Gary McGraw. Exploiting Software: How to Break Code. Pearson Higher Education, 2004.

[7] Michael Howard and Steve Lipner. The Security Development Lifecycle. Microsoft Press, Redmond, WA, USA, 2006.

[8] Thomas Maller, Rex Black, Sigrid Eldh, Dorothy Graham, Klaus Olsen, Maaret Pyhajarvi, Geoff Thompson, Erik van Veendendal, and Debra Friedenberg. Certified tester—Foundation level syllabus. On-line article, April 2007.

[9] Barton P. Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of UNIX utilities. Communications of the ACM, 33(12):32-44, 1990.

[10] David Thiel. Exposing vulnerabilities in media software. Black Hat conference presentation, BlackHat EU 2008.

[11] Jacob West. How I learned to stop fuzzing and find more bugs. Defcon conference presentation, August 2007.