Automated Testing
With Commercial Fuzzing Tools

A Study of Commercially Available Fuzzers: Identification of Undisclosed Vulnerabilities with the Aid of Commercial Fuzzing Tools

Prof. Dr. Hartmut Pohl and Daniel Baier, B.Sc.
Department of Computer Sciences,
Bonn-Rhein-Sieg University of Applied Sciences

This report is based on the principles and results from a research and development project funded by the German Federal Ministry of Education and Research and carried out at the Bonn-Rhine-Sieg University of Applied Sciences.
1. Summary

Bugs relevant to security in applications (vulnerabilities) are among the most frequent and thus riskiest attack targets in company IT systems. Cost-effective, tool-based Fuzzing techniques help to identify hitherto unknown security relevant bugs. The aim of this report is to analyze, assess and compare Fuzzing tools.

In a series of projects, hitherto unknown vulnerabilities in individual and standard software were identified and also fixed by the respective software developer using Fuzzing techniques. This report is aimed at comparing the efficiency of commercially available Fuzzers.

The good results achieved to date show that Fuzzing techniques identify critical vulnerabilities which are exploitable from the Internet - despite a high security standard in the programming guidelines [Pohl 2010a]. Commercial Fuzzers - beSTORM in particular - enable the quick and targeted examination of an application with respect to its security level because they can be operated intuitively and provide comprehensive interface support.

The efficiency of the individual techniques and tools was examined in practice - the Top 25 vulnerabilities can (only) be identified using a combination of Threat Modeling, Fuzzing and examining the source code [Pohl 2010b, 2010c; MITRE 2010].

2. Motivating Factors/Motivation

It is impossible to develop bug-free software. This makes the testing of software necessary, while maintaining a close link to quality assurance. It is impractical to conduct manual tests of software that has large quantities of programming codes [Sutton 2007]. Therefore, tool-based, automated techniques are required to identify vulnerabilities.

This is because security vulnerabilities in applications are among the most frequently discovered and thus riskiest attack targets in company IT systems. Vulnerabilities in applications enable, among other things, attacks from the Internet and thus data loss, (industrial) espionage, and sabotage. In many cases, these attacks also make it possible to successfully attack the underlying, internally linked company IT systems and thus to access company-internal IT networking systems and critical IT infrastructures, such as financial data, Enterprise Resource Planning Systems (ERP), Customer Relationship Management Systems (CRM), production control systems, etc.

Traditional security tools only enable the identification of known attacks that exploit known vulnerabilities; such security tools include, for instance, (web application) firewalls, intrusion detection systems, etc. Hence, they cannot be used to detect new attacks and new types of attacks.

Beyond this, state-of-the-art security tools support the identification of hitherto unknown vulnerabilities and the detection of new types of attacks based on these vulnerabilities.

- Threat Modeling makes it possible to identify vulnerabilities early in the design phase.
- Static source code analysis (Static Analysis) is aimed at analyzing the source code without executing it. This tests whether the code conforms to the programming language and the programming guidelines during the implementation phase. Static Analysis tools work like parsers that conduct lexical, syntactic and semantic analyses of programming codes.
- Dynamic analysis tools (Fuzzers) transmit random input data to the target program to trigger anomalous program behavior. Such anomalies indicate vulnerabilities.
Microsoft has been using the advantages of Threat Modeling [Howard 2006] and Fuzzing [Godefroid 2008] since 2003 as an integral part of its own "secure" software development process - the Security Development Lifecycle (SDL) [Howard 2006].

The cost of fixing vulnerabilities rises exponentially in the course of the software development process [NIST 2002]. If bugs are identified in the testing or verification phase, the cost rises by the factor 15 compared to their being detected during the design phase. If the bugs are identified during the release phase (or even later), the cost rises by the factor 100 (cf. Figure 1).

![Figure 1: Cost of Bug Elimination in the Software Development Lifecycle [NIST 2002]](image)

3. Fuzzing

3.1. Introduction to Fuzzing

Fuzz-testing (Fuzzing) is a software testing technique that is ideally used during the verification phase within the Security Development Lifecycle (SDL) [Lipner 2005], yet it is equally successful at a later stage, when the software has been delivered to the customer. The verification phase is located between the implementation and the release phase within the SDL (Figure 1).

The Fuzzing process [cf. Figure 2] describes how Fuzzing tests are conducted:

1. Identifying input interfaces,
2. generating input data,
3. transmitting input data,
4. monitoring the target software,
5. conducting an exception analysis and
6. drawing up reports/reporting.
After the interfaces have been successfully identified, input data can be generated using a Fuzzer; these data can then be transmitted to the target software to be tested. During the Fuzzing process the software is monitored so as to detect anomalous program behavior, which are triggered by randomly and intelligently selecting the widest possible range of input data.

![Fuzzing Process Diagram](image)

**Figure 2: Fuzzing Process**

At last the results are reported to management and the technicians; the latter compile all information found during the execution of the program.

Fuzzing is a tool-based technique used to identify software bugs during the verification phase; this can contribute to identifying undisclosed security relevant bugs. To this end, the input interfaces of the target software to be tested are identified, to which targeted data are directed in an automated fashion while the software is being monitored for potential bugs. This makes it possible to prevent third parties from identifying vulnerabilities and thus from developing zero-day-exploits [Pohl 2007].

Zero-day exploits are one of the twenty most frequent types of attacks [SANS 2010].

Fuzzing can be conducted both in the form of a white-box test (with available source code) and, above all, in the form of a black-box test (with no available source code) during the verification phase.

### 3.2. Market Analysis

There are more than 250 Fuzzers. 25% of all Fuzzers can be used to test web applications and 45% can be used to examine network protocols. The testing of file formats is supported by 15% of all Fuzzers. Web browsers can be examined by 10% of all Fuzzers, whereas APIs can be tested by 7% of all Fuzzing tools. There are only two multi-protocol, environment variable fuzzers. [Cf. Figure 3: Market Overview of Fuzzers]
3.3. Case Study: Testing Fuzzers

In a fuzzer case study done for Bonn-Rhine-Sieg University of Applied Sciences, a large number of vulnerabilities were identified using the commercial Fuzzers evaluated below. In this real world evaluation, published software, i.e. software applications purchasable on the market, was tested to failure. The levels of severity assigned to the vulnerabilities were calculated using the Common Vulnerability Scoring System [Mell 2007] and then graphically represented as "Critical", "High", "Medium", "Low", "Undefined" - according to their degrees of criticality. The more severe the vulnerability, the higher the potential harm and the lower the effort required to exploit the vulnerability. The vulnerabilities detailed in figure 4 were discovered during the testing process.

Figure 3: Market Overview of Fuzzers

Figure 4: Vulnerabilities Identified with the Aid of Fuzzing, during the Case Study
3.4. Fuzzer Evaluation Parameters

Various parameters have been drawn up to evaluate testing tools. The following eight evaluation parameters are important with respect to Fuzzing:

- **Supported Fuzzing techniques and protocols**: The aim of this parameter is to evaluate the possibility of using a Fuzzer to perform diverse tasks, including, for instance, the tool's ability to independently interpret interfaces, to adapt to protocol specifications and the scope for using it to interpret the target software.

- **Costs and license**: evaluation of the costs arising from each tool, including purchasing, maintenance and personnel costs as quantified on the basis of actual use.

- **Analytical abilities**: evaluation of the extent to which the Fuzzer is able to conduct analyses. This includes the monitoring techniques supported, the identification of vulnerabilities, the way the target software is reset and reporting. Furthermore, such criteria as the ability to establish parameters, bug reproduction, support for parallel Fuzzing as well as interruption and resumption of Fuzz tests are taken into account.

- **Operating systems**: this examines the question of which operating systems the Fuzzer can be used on and, above all, the question of whether the software is independent of the platform used.

- **Software ergonomics**: evaluation of the respective tool's efficiency, profitability and user-friendliness during the conduction of Fuzzing tests. Moreover, functional, dialog as well as input and output criteria are assessed.

- **Documentation**: evaluation of the completeness and quality of the documentation resources provided, such as user manuals, technical and third party documentation, as well as evaluation of the quality of the user interface.

- **Extendibility**: evaluation of the ability of the tool to supplement or extend existing features. The interfaces included, the programming language and the developing tools required are also taken into account.

- **Further parameters**: evaluation of further methods and features provided by Fuzzers to improve the quality of Fuzzing. Above all, the scope for identifying, defining, evaluating and presenting bugs is evaluated.

These parameters enable the consistent classification and evaluation of Fuzzers and serve as a basis of ranking them on their merits.

3.5. Fuzzer Analysis

In the following review, six different examples of commercial, widely used Fuzzers were examined. The Fuzzers examined and evaluated can be seen from Figure 5 -Commercial Fuzzers Examined.

Fuzzing tools from the category of "Multi-Protocol Fuzzers" support most protocols and can thus be used to examine several interfaces. Fuzzers from the category of "Web Application Fuzzers" count among the remote Fuzzers, even though their application is not restricted to web applications.
On the whole, Tool A stands out. It shows high standards of user-friendly handling and operation, of the Fuzzing techniques supported and the analytical abilities provided.

Both Tool A as well as Tool B enables the user to reset the target application after system failure and implements the reproduction of bugs identified. Tool A generates a Perl script that reproduces the bug (the vulnerability).

Tool B, which shows markedly higher purchasing costs, supports a larger number of Fuzzing techniques. However, the utilization of the individual features is made difficult by user interface complexity.

Tools C and F attain very good results for user friendliness. On the other hand, they both achieve low scores for their application possibilities and the Fuzzing techniques they support. The difference between them is very slight, with Tool F being slightly better than Tool C.

Tool D attains a satisfactory result for user friendliness. On the other hand, it excels in terms of its low purchasing costs. The tool only supports a small number of Fuzzing techniques, yet its analytical abilities are comprehensive.

Tool E excels in terms of very high scores for user friendliness. However, it only supports a small number of Fuzzing techniques, which are evaluated on the basis of the good analytical abilities it provides. Documentation is also assigned a low score.

The costs of the tools differ considerably. Tool B shows the highest purchasing costs. Tools A, C and F are in the same price segment, differing from each other only slightly. Tools D and E excel in terms of their low purchasing costs, with Tool D being considerably cheaper. Owing to their complexity and low user friendliness, Tools B and D are characterized by higher personnel costs.

Each of the evaluation parameters is assigned between 0 and a maximum of 10 points, with 10 points being the maximum, i.e. best (evaluation) result achievable. These results are graphically represented in Figure 6: Evaluation of the Fuzzers.
If Fuzzers are to be used for several interfaces, the degree to which interfaces are supported should be taken into account. The expertise required to use Fuzzers is another criterion on which the evaluation of Fuzzers may be based.

<table>
<thead>
<tr>
<th>Parameter Tool</th>
<th>Software Ergonomics</th>
<th>Documentation</th>
<th>Compatible Protocols</th>
<th>Analysis Depth</th>
<th>Further Company-Specific Parameters</th>
<th>Extendibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>+++</td>
<td>++</td>
<td>+++</td>
<td>+++</td>
<td>++</td>
<td>++</td>
</tr>
<tr>
<td>B</td>
<td>+</td>
<td>++</td>
<td>+++</td>
<td>+++</td>
<td>+</td>
<td>+++</td>
</tr>
<tr>
<td>C</td>
<td>+++</td>
<td>++</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>D</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>++</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>E</td>
<td>++</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>F</td>
<td>+++</td>
<td>+++</td>
<td>+</td>
<td>++</td>
<td>+</td>
<td>+</td>
</tr>
</tbody>
</table>

Figure 6: Evaluation of the Fuzzers

Figure 7: Comparison of Fuzzers on the Basis of the Interfaces and Expertise Required
Tools A and B support a larger number of interfaces than Tools C, D, E and F. However, Tool B requires a higher level of expertise. Cf. Figure 8: Comparison of Fuzzers on the Basis of the Interfaces and Expertise Required.

Figure 8: Evaluation Results: Comparison of Fuzzers on the Basis of the Interfaces and Expertise Required

4. Commercial Fuzzer Comparison Results

After comparing commercially available Fuzzers, beSTORM was selected as the overall best in class as it achieved above average results in all parameters. Apart from being particularly user-friendly in terms of handling, beSTORM excels in terms of the Fuzzing techniques supported and the analytical abilities provided; it supports 54 protocols and file formats.

beSTORM is a "Smart-Stateful-Grammar-Based-Generation-Based Fuzzer". It also contains a component to adapt to protocol specifications; hence, it can just as well be classified as a "Smart-Stateless-Grammar-Based-Model-Interference-Based Fuzzer". Furthermore, it can also be regarded as a "Protocol-Specific - Multi-Protocol Fuzzer" because it supports multiple protocols.

beSTORM is provided with the functionality to detect the occurrence of service denials - for instance, on the basis of such criteria as CPU activity, storage utilization and target program failure. The tool is also capable of identifying memory access violations.

beSTORM is highly efficient and cost-effective in terms of application; the dialogs meet expectations and are self-explanatory throughout. The basic usability criteria are complied with; even though the tool is particularly user-friendly in terms of handling, there is still enough potential for the future development of further versions.

The beSTORM user manual is available in English. It also contains a description of the protocol specification format and can thus be seen as technical documentation. A support system is also integral to the manual.
Along with the software, the purchaser receives a tutorial in the form of a quick start guide. The developer’s site provides the opportunity to gain insights into a large number of white papers and case studies. Webinars are also conducted.

In summary, beSTORM shows higher standards in a number of areas than its commercial competitors: above all, beSTORM excels in terms of the good software ergonomics it provides. The dialogs are user-friendly and self-explanatory; there is complete and detailed documentation available. The tool supports comprehensive Fuzzing techniques and has considerable analytical abilities.

5. References


