Improving Security Testing With Usage-Based Fuzz Testing

Martin A. Schneider[1], Steffen Herbold[2], Marc-Florian Wendland[1], and Jens Grabowski[2]

[1] Fraunhofer FOKUS, Berlin, Germany,
{martin.schneider,marc-florian.wendland}@fokus.fraunhofer.de
[2] Institute of Computer Science
University of Göttingen
Göttingen, Germany
{herbold,grabowksi}@cs.uni-goettingen.de

Abstract. Along with the increasing importance of software systems for our daily life, attacks on these systems may have a critical impact. Since the number of attacks and their effects increases the more systems are connected, the secure operation of IT systems becomes a fundamental property. In the future, this importance will increase, due to the rise of systems that are directly connected to our environment, e.g., cyber-physical systems and the Internet of Things. Therefore, it is inevitable to find and fix security-relevant weaknesses as fast as possible. However, established automated security testing techniques such as fuzzing require significant computational effort. In this paper, we propose an approach to combine security testing with usage-based testing in order to increase the efficiency of security testing. The main idea behind our approach is to utilize that little tested parts of a system have a higher probability of containing security-relevant weaknesses than well tested parts. Since the execution of a system by users can also be to some degree being seen as testing, our approach plans to focus the fuzzing efforts such that little used functionality and/or input data are generated. This way, fuzzing is targeted on weakness-prone areas which in turn should improve the efficiency of the security testing.

Keywords: Security Testing; Fuzzing; Usage-Based Testing

1 Introduction

Security testing is about finding potential security-relevant weaknesses in the interface of a System Under Test (SUT). In the last decade, vulnerabilities and their exploitation by hacker groups, industrial competitors, and adversary intelligence services became part of our daily lives. This makes it inevitable to detect and fix security-relevant weaknesses as fast as possible. This need will gain much more importance in the future due to the increasing connectivity of systems with real-world entities, e.g., with the emergence of cyber-physical systems and the

Internet of Things. The effort required for the security testing with currently established techniques increases dramatically with the complexity of the systems. To cope with this problem and to meet the higher requirements with respect to system quality, in particular security, the existing techniques have to become more efficient than they currently are and new techniques have to be devised. In this paper, we want to discuss a new approach based on the combination of fuzzing and usage-based testing in order to provide an automated way yielding an improved efficiency of security testing.

2 Related Work

Since we are combining several existing techniques, we present the chosen and alternative techniques.

2.1 Risk Analysis Appraches

There are several approaches aiming at identifying and assessing risks for certain failures. Fault-Tree Analysis (FTA) [1] is a top down approach. The analysis starts from an undesired state, subsequently exploring different faults and their interrelationships being expressed using logical gates. FTA enables both, qualitative and quantitative analysis.

In contrast to FTA, Failure Mode and Effects Analysis (FMEA) [2] is usually performed as a bottom up approach. Therefore, it does not start from an undesired state but from a malfunctioning component. Thus, the consequences of a component error are analyzed. If a criticality analysis is performed afterwards, it is called Failure Model Effects and Criticality Analysis (FMECA). As FTA, FMEA/FMECA enables qualitative and quantitative analysis.

Attack trees [3] are an approach with some similarity to FTA but fitted to the analysis of security risks. It takes into account the capabilities of an attacker and starts with the goal of an attack as a root node of the tree. The leafs constitute the attacks in order to achieve the goal connected via logical nodes. In additional to FTA, countermeasures can be included in the nodes.

CORAS [4] is an approach for model-based risk assessment, in particular used for security risk analysis. Whereas the aforementioned approaches are based on trees, during the risk analysis according to the CORAS method, graphs are created. The CORAS method comprises eight steps that lead to different kind of diagrams. The approach starts with the analysis of the assets wort protecting, followed by threat identification and estimation and the identification of treatments. CORAS diagrams provide different kind of nodes for threats, e.g., attackers, threat scenarios, vulnerabilities, unwanted incidents, i.e. the result of a successful attack, and considers likelihoods of, e.g., threat scenarios, and impacts on the assets.

All the presented approaches for risk analysis have in common the need for manual analysis performed by system and domain experts, which may require substantial effort.

2.2 Fuzzing

An established technique for finding security weaknesses is fuzzing [5]. Fuzzing is performed in an automated manner and means to stimulate the interface of a SUT with invalid or unexpected inputs. Therefore, it is a negative testing approach. Fuzzing aims at finding missing or faulty input validation mechanisms. This may lead to security-relevant weaknesses if such data is processed instead of being rejected. For instance, a buffer overflow vulnerability usually results from the lack of a length check of user input data. Thus, arbitrary long data can overwrite existing data in system memory and an attacker may use this for code injection. Due to the huge size of the input space, fuzzing research focuses on approaches of how to sample input data in a way that the likelihood of finding weaknesses is high. The existing approaches discussed cover randomly generated data [6] and model-based approaches where models and grammars describe valid and/or invalid input data [5]. Model-based fuzzers have knowledge about the protocol of the interface they are stimulating and are able to generate so-called semi-valid input data. This is data that is mostly valid but invalid in small parts. This allows checking the input validation mechanisms one after another. In order to detect a buffer overflow vulnerability, input data of invalid length would be generated in order to check if the length of input data is verified. However, even with such model-based techniques, the total number of generated inputs usually cannot be tested exhaustively due to time and resource limitations.

Recently, behavioral models have also been considered for fuzzing [7]. With behavioral fuzzing, invalid message sequences are generated instead of invalid data. For example, this can lead to finding weaknesses in authentication mechanisms, where the exact order of messages is relevant. However, this procedure also leads to a huge number of behavioral variations where executing all of them is usually infeasible due to a lack of time and resources.

As described above, the number of invalid inputs generated by fuzzing techniques is usually too large to execute all as tests. The challenge is to select those test cases that have a high probability of finding a weakness. While there are approaches to cope with this manually, e.g., risk-based security testing [8], the automated selection is still an unresolved research topic.

2.3 Usage-Based Testing

Usage-based testing is an approach for software testing that focuses the usage of a system. Instead of testing all parts of the system equally, the parts that are often used are tested intensively, while seldom or never used parts are ignored [9]. The foundation for usage-based testing are usage profiles, i.e., stochastic descriptions of the user behavior, usually as some form of Markov process (e.g., [10]). The usage profile is inferred from a usage journal which is collected by a monitor observing the SUT during its operation. The journal also contains the data users sent. Sensitive user data, e.g., names and passwords, are filtered from this data. This can be achieved by ignoring the values of certain observed elements, e.g., password fields or the tagging of fields that contain sensitive data, e.g., name fields.

3 Our Approach towards Usage-based Fuzz Testing

In this paper, we want to discuss a new approach for fuzzing that combines it with usage-based testing. The new approach shall resolve some of the efficiency problems of existing fuzzing techniques. The underlying assumption of our approach is that system execution of the users unveils faults, similar to functional testing. Therefore, most remaining faults in a system should be located in parts of the system that are not regularly executed by the users and thus, little tested. From this, we conclude that the same should be true for security-relevant bugs. Normally, usage-based testing generates test cases for the most used parts of a system, in terms of both functionality and data. For the purpose of finding security weaknesses, we plan to invert this approach: aim the testing at seldom used functionality with rarely used data.

In our approach, we consider both data and behavioral fuzzing to perform security testing. However, the information provided by a usage profile is utilized by both fuzzing techniques differently.

3.1 Preparing the Usage Profile

As discussed, we presume a negative correlation between tested functionality and risk for a security relevant bugs. Therefore, the probabilities within the usage profile have to be inverted and normalized, which automatically means that the focus is put on rarely used functionality. However, functionality that is never used is not considered. It is required to map the usage profile to a model of the SUT, e.g., an environmental model. This allows identifying the unused functionality having the highest risk for security relevant bugs according to our assumption.

3.2 Usage-based Data Fuzzing

For data fuzzing, the usage intensity of the inputs is of interest. We target fields where the inputs rarely varied, i.e. the usage profile provided only a small number of different or even no inputs. The probabilities provided by the usage profile can guide both the generation of a test scenario where the functionality is used as well as fuzz test data generation itself by utilizing information which values are already used by the users.

In addition, the different user inputs obtained from the usage profile can be considered in more detail. Users do not always provide valid input data for several reasons. This could happen due to mistypes, insufficiently educated users or unclear information what input is expected. The probabilities for events representing invalid data may be more reduced than those for events only representing valid user data and thus, reduce the chance that test scenarios are generated that are already covered by regular usage-based test cases that already using invalid input data. Therefore, if the same number of different input data is contained in the usage profile for a certain functionality, the events identifying invalid input would reduce the probability for corresponding test scenario generation more than events identifying valid input.

Through usage-based fuzz testing, we focus the fuzzing on seldom or never used system parts. We have two advantages in comparison to fuzzing without usage information. First, we reduce the risk of vulnerabilities in areas that are often neglected, first of all by the users, but as a side effect also by the maintenance of the software due to regular usage-based testing. Second, we reduce the computational effort for fuzzing because it is targeted on and restricted to areas that are likely to contain vulnerabilities.

3.3 Usage-based Behavioral Fuzzing

In contrast, for the behavioral fuzzing only the usage frequency of functionality is of interest. Here, we invert the probabilities of the usage frequency, in order to fuzz the behavior around rarely used functionalities by modifying the test scenarios on message level by applying dedicated behavioral fuzzing operators [7].

4 Advantages of the Proposed Approach

We expect from the proposed approach of usage-based fuzz testing advantages with respect to two aspects:

– Reduced effort for maintenance between minor versions with respect to security testing. Between different versions of a software, usage-based testing can be employed in order to ensure that the quality of the most used functionalities achieve at a certain quality level. In addition, security testing performed with usage-based fuzz testing is complementary in terms of usage frequency because the probabilities of the usage profile are inverted. Thus, the effort to security testing can be reduced by focusing on parts that weren't intensely tested having a high risk of security-relevant bugs.
– The more significant aspect would be the advantage resulting from the degree of automation. As discussed in Section 2, the existing approaches for risk analysis require substantial manual effort. The approach of usage-based testing considers seldom or never used and thus, little or even not tested functionality as risk. The usage profile is obtained automatically by a usage monitor. The subsequent steps, i.e. inverting the usage profile, mapping it to a model of the SUT, considering the provided inputs with respect to validity, can also be performed automatically. As a result, the approach is completely automated in contrast to other approaches that combine security testing with risk assessment.

5 Example

The approach of usage-based fuzz testing is illustrated in the following, based on an example of a simple input dialog as depicted in Figure 1. The input dialog consists of an edit box, an OK button, a cancel button, and a reset button that sets the value to the default value. The input box provides a default value that users may change by clicking in the input field and changing the default value to the desired one.

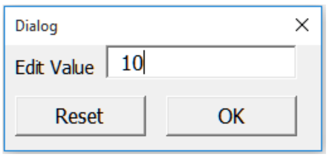

Fig. 1. Input Dialog

The events that can be observed when the dialog box is opened are 'Click Edit box', 'Type Into Edit Box', 'Click OK', 'Click Reset', and 'Click X'. Additionally, two events called 'Start' and 'End' are edit that represent the events that the input dialog appears and disappears. The edges between these events represent the probabilities that the destination of an edge occurs when the source event of the edge has occurred.

Figure 2 (a) depicts the events and the probabilities based on the observed event frequencies. It should be noted that most of the users do not change the given values. In only 10 % of the cases the default value is changed by typing once in the edit box. Changes including two or more changes are even rarer.

If there are functionalities that wasn't used in no cases, these can be added by mapping the usage profile to a model of the SUT. In this example, the event 'Click Reset' was never observed, hence, it is not contained in the usage profile. By mapping the usage profile to the a model of the SUT, the missing events and corresponding edges are added. They are depicted by dashed lines in Figure 2. The incoming and outgoing edges of the event 'Click Reset' that was not observed is therefore set to 0 %.

Given this usage profile, the one for usage-based fuzz testing can be derived by inverting the probabilities and normalizing them. The resulting usage profile is depicted in Figure 2 (b).

Based on the inverted usage-profile, test scenarios can be generated, e.g., by random walks. Due to the inverted probabilities of the usage profile, seldom used functionality is taken into account more intensely that frequently used functionality during test scenario generation. In the given example, test scenarios would be generated where a lot of input to the edit box would be made. Through regular usage-based testing, only few different test cases would cover this functionality. Therefore, possible faults may be missed. This might pose a security risk if input validation mechanisms are incorrectly implemented or missing. Considering a functional fault detected by a functional test case. In order to fix this bug, a developer would usually perform a review of the corresponding code snippet and thus, may discover also other faults in this area and fix them. Given that

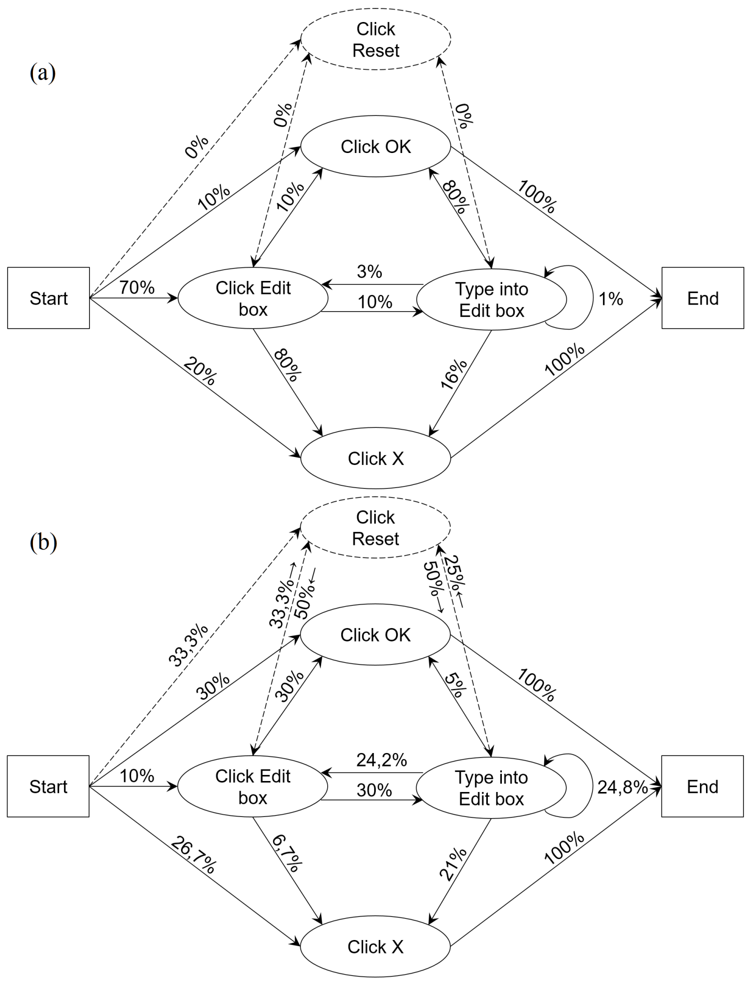

Fig. 2. (a) Usage profile including probabilities based on usage frequencies. Events that were not observed are added by mapping the usage profile to a model of the SUT. The additions were marked by dashed lines. (b) Mapped usage profile with inverted and normalized probabilities. It serves as a starting point for usage-based fuzz testing.

a user provided invalid data, the developer would review the validation mechanism and thus, would probably find other security-relevant faults. Therefore, parts that are subject to usage-based testing may have a reduced risk to faults with respect to input validation.

Considering the seldom used functionality of changing the default value, according the usage-based testing approach, only few test cases would be generated that would cover few different inputs to the edit box. Therefore, existing faults are unlikely being detected and this is where the approach of usage-based fuzz testing comes into play. Resulting from the usage profile with inverted probabilities, test scenarios are generated that cover the typing into the edit box as depicted in Figure 3. Usage-based data fuzzing will generate many test cases that create different malicious inputs submitted to the edit box (grey shaded in Figure 3). Therefore, different kinds of injection vulnerabilities may be discovered having a higher chance of success due to the small number of usage-based test cases. Considering the invalid inputs provided by the usage profiles, usage-based data fuzzing is able to focus on such possible faults that were neglected by usage-based testing due to a small usage frequency. Invalid user inputs may range from values that might be out of certain range, i.e. too large or too small with respect to numbers. Those may be considered by reducing probabilities within the inverted usage profile. Utilizing this information from the usage-profile, malicious inputs covering other kind of vulnerabilities are targeted that are rarely subject of user input, such as SQL injection. The event 'Type into Edit Box' seldom occurring, therefore, usage-based data fuzzing would focus on this field, and would neglect other field even more intensely used by users. Eventually, these test cases are supplemented by those generated by usage-based behavioral fuzzing aiming at the discovery of functionality that should be disallowed.

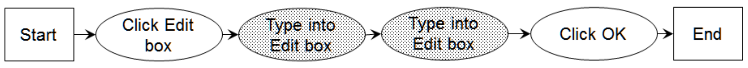

Fig. 3. A test scenario generated from the inverted usage profile. The grey parts are the target of fuzz test data generated based on the information from the usage profile.

6 Evaluation within the MIDAS Project

Within the MIDAS European research project [11] we are currently building a test platform on the cloud for testing of Service Oriented Architectures (SOAs). Figure 4 provides an overview of the MIDAS Platform. As part of this project, we are implementing data fuzzing, behavioral fuzzing, and usage-based testing all on the same input model. Our joint input model is a Domain Specific Language (DSL) based on Unified Modeling Language (UML) and UML Testing Profile (UTP), which already provides mechanisms for defining fuzzing operators for test cases. Moreover, the tooling for the creation of a usage profile is also developed as part of MIDAS, including usage monitoring facilities for SOA applications. The usage-based testing facilities provide the functionality to generate test cases compliant to the DSL, which can then be extended with the appropriate fuzzing operators. The presented approach of usage-based fuzz testing is achieved by an orchestration of the MIDAS test generation services for usage-based testing and fuzz testing. The tool developed for usage-based testing is called AutoQUEST [12] developed by the University of Göttingen and is integrated and improved for the MDIAS Platform. Data fuzzing within the MIDAS Platform is based on the fuzz test data generator Fuzzino [13] extended for testing web services.

Within the project, we are working together with industrial partners who supply us with both usage data as well as systems where security is a relevant property. The first system we consider comes from the health care domain and is concerned with the management of patient data, i.e., sensitive data that must be protected against adversaries. The second system we use to evaluate is a supply chain management system, where security leaks can lead to wrong orders and manipulation of databases, which can costs a lot of money, depending on the industry they are used in.

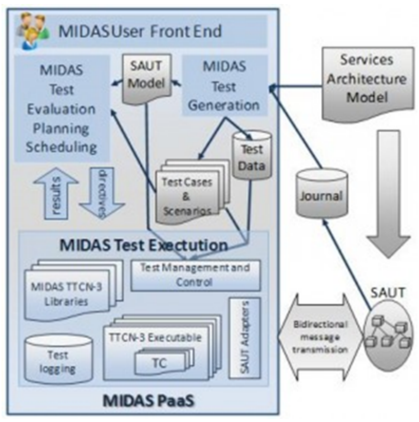

Fig. 4. Overview of the MIDAS Platform.

7 Conclusion and Future Work

We propose the idea of usage-based fuzz testing, an approach that focuses data and behavioral fuzzing on rarely used and thus, little tested parts of a software. With our work in the MIDAS project, we already built a strong foundation for the implementation of such an approach. In the future, we will investigate how to best combine these techniques in order to leverage the strengths of both approaches. These investigations will include how information about data usage from the usage profile can be used to guide data fuzzing and how the test cases derived by usage-based testing can serve as foundation for behavioral fuzzing.

Acknowledgment

This work was partially funded by the EU FP 7 projects MIDAS (no. 318786) and RASEN (no. 316853).

References

1. I. E. Commission, “Iec 61025 fault tree analysis,” 1990.
2. ——, “Iec 60812 analysis techniques for system reliability-procedure for failure mode and effects analysis (fmea),” 2006.
3. B. Schneier, “Attack trees,” *Dr. Dobbs journal*, vol. 24, no. 12, pp. 21–29, 1999.
4. M. S. Lund, B. Solhaug, and K. Stølen, *Model-driven risk analysis: the CORAS approach*. Springer Science & Business Media, 2010.
5. A. Takanen, J. DeMott, and C. Miller, *Fuzzing for software security testing and quality assurance*, ser. Artech House information security and privacy series. Artech House, 2008. [Online]. Available: http://books.google.de/books?id=tMuAc_y9dFYC
6. B. P. Miller, L. Fredriksen, and B. So, “An empirical study of the reliability of UNIX utilities,” in *In Proceedings of the Workshop of Parallel and Distributed Debugging*. Academic Medicine, 1990, pp. ix–xxi,.
7. M. Schneider, J. Grossmann, N. Tcholtchev, I. Schieferdecker, and A. Pietschker, “Behavioral fuzzing operators for uml sequence diagrams,” in *System Analysis and Modeling: Theory and Practice*, ser. Lecture Notes in Computer Science, y. Haugen, R. Reed, and R. Gotzhein, Eds. Springer Berlin Heidelberg, 2013, vol. 7744, pp. 88–104. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-36757-1_6
8. “EC FP7 RASEN Project,” www.rasenproject.eu, 2012-2015, FP7-316853.
9. S. Herbold, “Usage-based Testing of Event-driven Software,” Ph.D. dissertation, Dissertation, Universität Göttingen (electronically published on http://webdoc.sub.gwdg.de/diss/2012/herbold/), June 2012.
10. P. Tonella and F. Ricca, “Statistical testing of web applications,” *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 16, no. 1-2, pp. 103–127, 2004.
11. “EC FP7 MIDAS Project,” www.midas-project.eu, 2012-2015, FP7-316853.
12. F. G. Steffen Herbold, Patrick Harms. (2014) Autoquest. [Online]. Available: https://autoquest.informatik.uni-goettingen.de/
13. M. Schneider. (2013) Fuzzino. [Online]. Available: https://github.com/fraunhoferfokus/Fuzzino