2012 Fourth International Conference on Multimedia Information Networking and Security

Software Vulnerability Discovery Techniques: A Survey

Bingchang Liu, Liang Shi†, Zhuhua Cai, Min Li
Software School of Xiamen University
Xiamen, China
e-mail: lbc19890812@yahoo.cn, sliang@xmu.edu.cn†

***Abstract*—Software vulnerabilities are the root cause of computer security problem. How people can quickly discover vulnerabilities existing in a certain software has always been the focus of information security field. This paper has done research on software vulnerability techniques, including static analysis, Fuzzing, penetration testing. Besides, the authors also take vulnerability discovery models as an example of software vulnerability analysis methods which go hand in hand with vulnerability discovery techniques. The ending part of the paper analyses the advantages and disadvantages of each technique introduced here and talks about the future direction of this field.**

Keywords-Vulnerability, Software static analysis, Fuzzing, Penetration testing, vulnerability discovery model

I. INTRODUCTION

There are many different definitions about software vulnerability. R. Shirey [1] defined software vulnerabilities as defects or weaknesses in system design, implement or operation management and can be used to break through security policies. Papers [2, 3, 4] respectively gave definitions of vulnerability from the perspectives of access control, state space and participants. A. Ozment thinks vulnerability research field is similar to software reliable engineering field, but the former is less mature than the latter. A. Ozment thinks one of the reasons is that the former lacks standard terminology definitions which exist in the latter. A. Ozment gives his own definition of vulnerability by reference to standard terminology definitions in software reliable engineering field: a software vulnerability is "an instance of a mistake [5] in the specification, development, or configuration of software such that its execution can violate the explicit or implicit security policy [6]. Firstly, this definition refers to some standard definitions that have been widely accepted in the software reliable engineering field, such as "mistake", which notes that a vulnerability is a single instance of mistake. So, there is no doubt whether two different instances of the same mistake are different vulnerabilities or not, to which the answer is very important for the research on independence of vulnerability discovery process. Secondly, it highlights different areas where a software vulnerability can originate: specification, development, configuration. Finally, it emphasizes the security policy rather than security system. So, the paper takes it as the definition of software vulnerability.

Research on software vulnerability can be classified into two categories: software vulnerability analysis and software vulnerability discovery. Software vulnerability analysis focuses on researching discovered software vulnerabilities to know characteristics of vulnerabilities, such as cause, position and implement features, and characteristics of vulnerability discovery process, such as features of vulnerability discovery rate. It aims to guide detection of other same kind vulnerabilities and new unknown vulnerabilities, and how to defend and avoid vulnerabilities of this kind. Software vulnerability discovery tries to discover existed but unexposed vulnerabilities, which is the focus of this paper. The key idea of vulnerability discovery is that the latter vulnerabilities are discovered the more expensively they are remedied [7]. So, how to discover software vulnerabilities has become the focus of vulnerability researchers. Software vulnerability discovery techniques have developed from original pure manual discovery to computer assisted discovery, and moves towards fully automated direction. Now, the main vulnerability discovery techniques include static analysis, dynamic analysis, Fuzzing and penetration testing. From the view of concept, the four techniques partially overlap, for example, Fuzzing belongs to dynamic analysis in essence and it has been introduced as the representative of dynamic analysis here. Besides, because results of software vulnerability analysis have the guiding significance for software vulnerability discovery, this paper also talks about vulnerability discovery models VDMs, which have become a significant branch of software vulnerability analysis.

The rest of this paper is organized as follows. Section 2 talks about static analysis; Section 3 talks about Fuzzing; Section 4 talks about penetration testing; Section 5 talks about VDMs; Section 6 analyses advantages and disadvantages of each technique and forecasts future direction of this field.

II. STATIC ANALYSIS

Static analysis is the process of evaluating a system or component based on its form, structure, content, or documentation, which does not require program execution [5]. The basic idea of static analysis is that it checks program text statically in order to discover defects or weaknesses in the program; especially those can lead to vulnerabilities. Manual code inspection is a traditional static analysis technique, but it requires spending lots of time and inspectors to have prior knowledge about the vulnerabilities to detect. Researchers come up with static analysis tools for that to perform static analysis with the aid of computer. However, static analysis tools do not make static analysis fully automated, whose outputs still need human to verify and estimate. Now, when talked about, static

This paper is Supported by the Fundamental Research Funds for the Central Universities(No. 2011121024)
† : Corresponding author

978-0-7695-4852-4/12 $26.00 © 2012 IEEE
DOI 10.1109/MINES.2012.202

analysis means that completed by static analysis tools in general and here is no different.

According to Rice theorem, any nontrivial problem about program can be reduced into halting problem [8]. So it is undecidable to decide whether there are still vulnerabilities in the program or not and discover all possible vulnerabilities in a program. That is to say, on one hand, some vulnerabilities in the program can not be found by static analysis forever [9]; on the other hand, static analysis can only approximate programs' behaviors. The former tells us we can not think the system to have no vulnerability when the output of static analysis is like that" no vulnerability found". The latter implicates results of static analysis are not perfect. False negative and false positive are two typical problems. Because static analysis is undecidable, false negative is strictly unavoidable. While impacts of false negative are more serious than false positive's, false positive is more disgusting. False positive requires human to verify and confirm the outputs of static analysis, which notes static analysis must work with human. Another characteristic of static analysis is its conservatism, meaning that its results are not precise enough and maybe not useful enough [10].

Based on targets' type, static analysis can be classified as static analysis for source code and static analysis for machine code. In theory, their nature and applicable method are the same, only, decoding the latter is much troublesome. Now, most of static analysis techniques are aimed at source code. There are many types of static analysis techniques which vary from simplify to complexity. Probably Unix utility grep is the simplest and most straightforward approach to static analysis. grep is based on the method of string matching and does not understand anything about the program it scans. However, it is armed with a list of good search strings and so it can expose many code problems. Methods based on lexical analysis are a little more complex. IST4 and RATs are such static analysis tools. These tools firstly pre-process and tokenize source files and then match the resulting token stream against a library of vulnerable constructs. Such methods produce numbers of false positive because they also do not understand the target code's semantics. But anyway they are better than methods of string matching. SWORD4J implement algorithm MARCO [11, 12] to check if java program scanned violates security policies of SBAC (Stack Based Access Control). MARCO first models the stack inspection mechanism on computed call graph G, then maps each node of G to a set of privileges with data-flow algorithm, and finally propagates a set of privileges necessary for current operation backward in G, until a fixed point is reached. Algorithm ESPE [11, 13] checks RBAC(Role Based Access Control) program's positions where RBAC security policies are violated by data-flow analysis. ESEP first computes the call graph G, then maps each node of G to a logical paradigm consisted of roles, and finally checks whether current role makes the corresponding node's logical paradigm true.

Static analysis can also be used to analyze information flow to verify information flow's integrity and confidentiality. The basic idea behind using static analysis for detecting information flow is to statically check that flow of information between variables in a program is consistent with the security labeling of variables [14]. Based on PDGs (Program Dependence Graphs) and non-interference, which dictates that low-security behavior of program can not be affected by any high-security data [15], Hammer et al. [16] presents a static analysis algorithm for information flow. The key idea of the algorithm is that if information flow is secure, for any statement s in the program, s' static backward slice can only contain statements that have a lower security label than s. Frame C [17] is a set of collaborative analyzers. It spans abstract syntax tree depending on C Intermediate Language CIL, performs value analysis for each program variable based on abstract interpretation, verify attributes by automatic theorem provers, divides program into small parts with program slicing and so on. Other interesting approaches include ESP [18] (a large-scale property verification approach) and model checkers such as BLAST [19].

In general, static analysis techniques include rule matching, data-flow analysis, control-flow analysis, program dependence analysis, information-flow analysis, static slice, abstract interpretation, model checking and theorem provers.

III. FUZZING

Inspired by Modem applications' tendency to fail due to random input caused by line noise on "fuzzy" telephone lines, in 1990, Miller et al. [20] proposed the concept of Fuzzing and developed the first Fuzzing tool "fuzz", which generates streams of random characters. Tested on ninety different utility programs on seven versions of UNIX, it was able to crash more than 24% of them. There are several definitions about Fuzzing in academia. Miller et al. [20] have defined Fuzzing as a randomized testing technique; P. Oehlert [21] have defined Fuzzing as —a highly automated testing technique that covers numerous boundary cases using invalid data (from files, network protocols, API calls, and other targets) as application input to better ensure the absence of exploitable vulnerabilities; M. Sutton et al. [22] regard Fuzzing as brute force testing. A. Lanzi et al. [23] have defined Fuzzing as a form of black-box testing. This paper will give definition about Fuzzing from the view of its procedure: firstly, it generates semi-valid or random data; secondly, it sends the generated data into the target application; finally, it observes the application to see if it fails as it consumes the data. Semi-valid data is data that is correct enough to pass input examinations, but still invalid enough to cause problems.

Data generation and target monitoring is the key to Fuzzing. In the early days, data was generated in a fully randomized way; as a result, most of the generated data was completely invalid. Later, researchers proposed two main methods of data generation, including data- generation technique and data-mutation technique [21]. Data-generation technique is usually based on specifications, such as file format specifications and network protocol specifications, to generate data. This technique requires that users have a lot of knowledge about the tested file formats and protocols, and needs much human involvement in the process. Data-mutation technique generates data by modifying some fields of valid inputs, which just requires users to have a little knowledge about file formats and protocols. When specifications are very complex and sample data is easy to collect, data-mutation is more appropriate than data-generation. However, data-mutation is not good enough for highly sensitive situations, and heavily depends on initial values, which means different initial values may lead to very different code coverage rate and effect [21]. Autodafe and APIKE Proxy are tools that generate data with data-mutation technique. Peach is a tool that combines the two techniques. With respect to target monitoring, most of tools use third-party debug tools, such as OllyDbg and GDB, while some other Fuzzing tools, such as Sullery and Autodafe, use their own custom debuggers. In addition to debugging target applications,

tools should also observe some other parameters, such as memory usage, CPU utilization and network activities [21, 24].

Talking about Fuzzing, many people will think of black-box testing, but it is not. P. Godefroid et al. [25] proposed white-box Fuzzing combining Fuzzing and dynamic test generation [26, 27]. Firstly, white-box Fuzzing constructs an initial and well-formed input I0, injects the input into program P, and Symbol execution observes P's processes on I0 and a path constraint that is in the form of logical formulas; secondly, it negates some marks of the path constraint, solves new constraint by a constraint solver, and create a new input I1 whose execution path is different from I0's; finally, it processes I1 in the same way with I0 and repeats the previous three procedures until time is out or no more new input is generated. The goal of white-box Fuzzing is to execute as many control paths as possible and discover vulnerabilities as fast as possible with all kinds of search strategies [28]. P. Godefroid et al. [28] introduced grammar describing valid input into white-box Fuzzing and made tokens returned from a lexical analyzer instead of the bytes in program inputs as symbolic. For applications with highly-structured inputs, this approach solves early path explosion problems and problems in which code beyond those first stages can not be reached. By experiments, P. Godefrorid et al. have found that white-box Fuzzing based on grammar is better than white-box Fuzzing and black-box Fuzzing; besides, white-box Fuzzing based on grammar is better than black-box Fuzzing based on grammar [28].

When detecting vulnerabilities with Fuzzing, people will encounter many problems. Validation for file formats and protocols is an obstacle to input generation. Some applications match key words with hash values, which has a negative effect on solving constraints. Besides, inputs may be processed with signature algorithms, encryption algorithms or compression algorithms. With respect to this, fuzzer should have the ability to re-sign, re-encrypt and re-compress.

IV. PENETRATION TESTING

Penetration testing, evaluates the security of a system by simulating attacks by malicious users and assessing whether the attacks are successful [29]. Penetration testing is more like an art than a science, because it does not depend on falsifiable hypotheses.

Penetration testing conducted by penetration testers employed by software suppliers is usually regarded as the final test before the software is published. According to the amount of information provided by suppliers, penetration testing can be divided into the following 3 categories:

1) Black-box penetration testing. It assumes no prior knowledge of the infrastructure to be tested. The testers must first determine the location and extent of the systems before commencing their analysis. Black-box penetration testing simulates an attack from someone who is unfamiliar with the system.

2) White-box penetration testing. It provides the testers with complete knowledge of the infrastructure to be tested, often including network diagrams, source code, and IP addressing information. White-box penetration testing simulates what might happen during an "inside job" or after a "leak" of sensitive information, where the attacker has access to source code, network layouts, and possibly even some passwords

3) Gray-box penetration testing. There are some variations between Black-box and White-box penetration, often known as Gray-box penetration testing.

When determine to conduct penetration testing, which one to select for software suppliers depends on suppliers' objectives [30].

In general, penetration testing consists of two processes: suppliers planning test and testers executing test. Test plan should build on risk analysis and threat modeling. For threat modeling, suppliers can make use of STRIDE [31], one of the best and most widely used methods, and the result can be presented in the form of Threat Trees [32]. Then based on the threats analyzed in the former steps suppliers make a test plan, often including penetration testing type (Black-box, White-box, or Gray-box), logistical arrangements, timetable, tools, requirements for the problem report and so on. With respect to testers executing test, PTES (Penetration Testing Execution Standard) [33] is recommended. PTES mainly consists of seven steps: a. Pre-engagement Interactions; b. Intelligence Gathering; c. Threat Modeling; d. Vulnerability Analysis; e. Exploitation; f. Post Exploitation; g. Reporting. It is worth mentioning that the submitted report should not just indicate if vulnerabilities exist in the target software. The report should contain more detail information, often include reproduction steps, severity, exploit scenarios and exploit code samples. Such reports are good for suppliers to evaluate the target's security risk and make a new publishment plan [34].

There are several things about penetration testing easy to be misunderstood. Firstly, some people may equate penetration testing with black-box testing, which leads to the issue of "security through obscurity" that means the defense is predicated on the attackers' lack of information. This happens to be the reason for many successful attacks [30]. Current best practices (e.g., OWASP [35] ， OSSTMM [36]) recommend that penetration testers assume that attackers have access to one or more versions of the source code of the application. Secondly, someone is trend to think the target software is immune to any attack after executing penetration testing. Rather, no vulnerability exposed or remedying all exposed vulnerabilities does not represent that no more vulnerability exists in the target, because the problem whether there are more vulnerabilities or not is undecidable. Finally, someone may think penetration testing immoral. In fact, penetration testing is just a tool to solve security problems and it is good or bad depending on the intent of its practitioner and of the recipient of its results [37]. About this issue, M. Bishop [32] recommended that sponsors write agreement with testers.

V. VDMS

O. Alhazmi and Y. Malaiya [38] applied Software Reliability Models (SRMs) to vulnerability discovery and called it Vulnerability Discovery Models (VDMs). O. Azment [6] defined that A vulnerability discovery model (VDM) specifies the general form of the dependence of the vulnerability discovery process on the principal factors that affect it: e.g. vulnerability introduction, vulnerability removal, detector effort, and the operational environment. VDMs model the vulnerability discovery process on vulnerabilities exposed in a certain software, in order to forecast the future vulnerability discovery process. Among the outputs of VDMs, the estimate of the total number of vulnerabilities and the mean time to next vulnerability (MTTNV) are two particularly estimates. The two

outputs can guide vulnerability detection process, such as resource allocation, personnel arrangements and timetable.

As VDMs derive from SRMs, many VDMs are intrinsically SRMs, which are applied to vulnerability discovery, rather than new models. E. Rescorla [39] applied a linear vulnerability discovery model (LVD) and an exponential Goel-Okumoto model to vulnerability data from NVD (National Vulnerability Database) and focused on investigating whether or not the social value of vulnerability discovery by extern detectors is positive. R. Gopalakrishna et al. [40] focused on discussing VDMs' theoretical and practical requirements. A. Ozment [41] tested the goodness-of-fit and predictive accuracy of several SRMs applied to vulnerability discovery. Besides, there are models specially designed for vulnerability discovery. The two models proposed by O. Alhazmi [38] are such ones: AME (Alhazmi-Malaiya Effort-based model) and AML (Alhazmi-Malaiya Logistic model). AME is an effort-based model and its basic assumption is that vulnerability detectors' effort is associated with the number of users of target systems. Experiments showed that AME had an acceptable chi square goodness-of-fit to Win98, WinNT4, Apache and IIS from NVD [38]. AML is an S-shaped, time-based logistical model of vulnerability discovery. AML assumes that vulnerability discovery takes place in three phases: a. learning phase. Detectors are still learning about the newly published software and do not report many vulnerabilities. This phase corresponds to the initial, flat portion of the S-shaped; b. linear phase. Detectors have become familiar with the software and the number of vulnerability reported grows linearly. This phase corresponds to the steep portion of the S-shaped; c. saturation phase. Users migrate to a newer version of software, detectors' interest become lower and the number of vulnerability reported decreases. Experiments showed that AML had an acceptable goodness-of-fit to vulnerability data from five Windows OS and two Linux OS. Later, O. Alhazmi et al [42] introduced vulnerability density to discard some of the extreme estimates produced by AML, named AML-C. Besides, some other researchers have placed their focus on assessing VDMs. Sung-Whan Woo et al. [43] assessed the predictive capacity of AME and AML by applying them to vulnerability data from NVD on Apache and IIS. Viet Hung Nguyen et al. [44] assessed the fitness of several VDMs to vulnerability data on browsers such as IE, Firefox, and Chrome.

To assess the effectiveness of a VDM, people need to test its goodness-of-fit and predictive accuracy including absolute predictive accuracy and relative predictive accuracy. An acceptable goodness-of-fit is a necessary condition for VDMs' effectiveness. Goodness-of-fit is generally tested with chi square. The VDM literature has relied upon two different approaches to assessing absolute predictive accuracy. One tests one-step-ahead predictive accuracy and the other ascertains the accuracy of the model's continuous predictions of the total number of vulnerabilities in the product. With respect to the relative accuracy of a model, two tests are AIC (Akaike Information Criterria) and PLR (Prequential Likelihood Ratio). AIC is an entropy-based approach used to find the model that best explains the data with a minimum of free parameters. PLR examines whether one model's probability density function is everywhere closer to the real probability density function than the other model being tested.

There are many problems about VDMs now. Most of VDMs are SRMs and inherit SRMs' assumptions. However, many assumptions of SRMs are wrong in vulnerability discovery process, which leads to uncertain effectiveness of these VDMs. VDMs face particular challenges in satisfying four assumptions: time, operational environment, independence, and static code.

VI. Comparison analysis and Future Direction

TABLE 1 TECHNIQUES COMPARISON

Technique	Advantages	Disadvantages
Static analysis	No requirements of executing target programs; Sound to describe properties of programs; easily integrated into the whole software development circle; Able to find most of implement bugs before the release of software.	Not precise enough to describe program properties; High false positive; Need human to verify the results and can not be entirely automatic; Most of such methods depend on source code; Unable to detect design bugs; Unable to detect vulnerabilities caused by configurations or environment.
Fuzzing	Simple idea; Easy to be understood; No false positive; High automation degree.	High randomness; High false negative; Low degree of generalization and long construction circle of Fuzzing tools.
Penetration Testing	No false positive and vulnerability discovery equal to vulnerability exploit; Based on practical user environments; Able to expose vulnerabilities hard to be detected by other tools; Take social engineering factors into consideration.	Heavily depend on human and the results depend to a great extent on testers' abilities, skills and experience; May do harm to the tested system.
VDMs	A new method to make use of the discovered vulnerabilities; In theory, able to predict the rate of vulnerability discovery and the total amount of vulnerabilities in a single software. Help to assess threats.	Some assumptions some VDMs base on need to be validated; Only apply to a single software; Lack general valid VDMs.

From table 1 we can see each technique has its own advantages and disadvantages. VDMs make use of the discovered vulnerabilities and its results help to assess threats that the target systems face with and provide three other techniques with direction in their design and implement. Static analysis can find most of the implement bugs in the development phrase. Fuzzing is generally conducted after development; it is still the technique which discovers the most vulnerabilities. Penetration testing is an art; it can find vulnerabilities hard to be exposed by tools; At the same time, it take social engineering factors into consideration. Because these techniques have different concerns, each of them needs to be studied further and the situation won't happen where one of them takes place of another.

Based on the requirements of vulnerability discovery, we talk about the future development of each technique as showed in table 2.

TABLE 2 THE FUTURE DIRECTION

Technique	Future
Static analysis	Static analysis based on machine code, including novel algorithms, tools development, platform's building and so on; Reduce false positive rate and involvement of human to improve the level of automation; Intelligent static analysis which can understand the program semantics and assess the

	discovered vulnerabilities. Combination with dynamic analysis.
Fuzzing	Improve the level of automation of knowledge acquisition, for example one can integrate automatic protocol analysis into Fuzzing; Make use of algorithms, such as simulated annealing algorithm and genetic algorithm, to avoid high randomness and high false positive rate; Data generation technique for multi-dimensions; Technique for assessing the effect of Fuzzing.
Penetration testing	Penetration testing education, mainly including training penetration testers; How to integrate penetration testing into each phrase of software development;
VDMs	Improve vulnerability database to provide VDMs with more detail information; Validate the un-validated assumptions; Develop novel VDMs; Assess the existed VDMs.

VII. CONCLUSION

This paper discusses vulnerability discovery from the view of engineering method, including static analysis, Fuzzing, penetration testing and VDMs. We have discussed each technique's concept, research status and related problems. Then, we analyze the advantages and disadvantages of each technique introduced here and talks about the future direction of this field

REFERENCES

[1] R. Shirey. Internet Security Glossary. RETF RFC2828, 2002

[2] Dorothy Elizabeth Robling Denning. Cryptography and Data security. 1982

[3] M. Bishop, D. Bailey. A Critical Analysis of Vulnerability Taxonomies, Technical Report CSE- 96-11. Dept. of Computer Science, University of California at Davis, Davis, CA 95616-8562, 1996

[4] Dennis Longley, Michael Shain. The Data & Computer Security 9Dictionary of Standards, Concept, and Terms, 1990

[5] IEEE. IEEE standard glossary of software engineering terminology. 1990

[6] Andy Ozment. Vulnerability Discovery & Software Security. University of Cambridge Computer Laboratory Computer Security Group &Magdalene College. 2007

[7] B. Boehm. Software Engineering Economics. 1984

[8] Brian Chess, Gary McGraw. Static Analysis for Security. IEEE Security and Privacy, Volume 2 Issue 6, 2004

[9] L. Osterweil. Integrating the Testing, Analysis, and Debugging of Programs. Proc. Symp. Software Validation, 1984.

[10] Michael D. Ernst. Static and dynamic analysis: synergy and duality. In WODA 2003: Workshop on Dynamic Analysis, (Portland, Oregon), 2003

[11] M. Pistoia. A Unified Mathematical Model for Stack and Role Based Authorization Systems. Ph.D. dissertation, Polytechnic University, Brooklyn, NY, USA, 2005

[12] M. Pistoia, R. J. Flynn, L. Koved, and V. C. Sreedhar. Interprocedural Analysis for Privileged Code Placement and Tainted Variable Detection. Proceedings of the 9th European Conference on Object-Oriented Programming, Glasgow, Scotland, UK, Springer-Verlag, 2005

[13] M. Pistoia, S. J. Fink, R. J. Flynn, and E. Yahav. When Role Models Have Flaws: Static Validation of Enterprise Security Policies. Proceedings of the International Conference on Software Engineering, Minneapolis, MN, USA, 2007

[14] M. Pistoia, S. Chandra, S. J. Fink, E. Yahav. A survey of static analysis methods for identifying security vulnerabilities in software systems. IBM SYSTEMS JOURNAL, VOL 46, NO 2, 2007

[15] J. A. Goguen, J. Meseguer. Security Policies and Security Models. Proceedings of the 1982 IEEE Symposium on Security and Privacy, Oakland, CA, USA, IEEE Computer Society Press, 1982

[16] C. Hammer, J. Krinke, and G. Snelting. Informati on Flow Control for Java Based on Path Conditions in Dependence Graphs. Proceedings of IEEE Internati onal Symposium on Secure Software Engineering, Arlington, Virginia, USA, 2006

[17] Pascal Cuoq et al. Experience report: OCaml for an industrial-strength static analysis framework. Proceedings of the 14th ACM SIGPLAN international conference on Functional programming, 2009

[18] M. Das, S. Lerner, and M. Seigle. ESP: Path-Sensitive Program Verification in Polynomial Time. Proc. ACM Conf. Programming Language Design and Implementation (PLDI2002), ACM Press, 2002

[19] T.A. Henzinger et al. Software Verification with Blast. Proc. 10th Int'l Workshop Model Checking of Software, LNCS 2648, Springer-Verlag, 2003

[20] Barton P. Miller, Louis Fredriksen, Bryan So. An empirical study of the reliability of UNIX utilities, Communications of the ACM 33, 1990

[21] Peter Oehlert. Violating Assumptions with Fuzzing. IEEE Security and Privacy, Volume 3 Issue 2, 2005

[22] M. Sutton, A. Greene, and P. Amini. Fuzzing: Brute Force Vulnerability Discovery. Addison-Wesley, 2007

[23] Andrea Lanzi, Lorenzo Martignoni, Mattia Monga, Roberto Paleari. A Smart Fuzzer for x86 Executables, Software Engineering for Secure Systems, 2007

[24] Leon Juranić. Using fuzzing to detect security vulnerabilities. INFIGO-TD-01-04-2006, 2006

[25] P. Godefroid, M. Levin, and D. Molnar. Automated whitebox fuzz testing. In NDSS, 2008.

[26] C. Cadar, V. Ganesh, P. Pawlowski, D. Dill, and D. Engler. EXE: automatically generating inputs of death. In CCS, 2006

[27] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In PLDI, 2005

[28] Patrice Godefroid, AdamKie˙zun, Michael Y. Lev -in. Grammar-based Whitebox Fuzzing. Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation, 2008

[29] William G.J. Halfond, Shauvik Roy Choudhary, and Alessandro Orso. Penetration Testing with Improved Input Vector Identification. 2009 International Conference on Software Testing Verification and Validation, EEE Computer Society Washington, DC, USA, 2009

[30] Matt Bishop. About Penetration Testing. IEEE Security & Privacy, 2007

[31] M. Howard and D. LeBlanc. Writing Secure Code. 2nd ed., Microsoft Press, 2002

[32] HERBERT H. THOMPSON. Application Penetration Testing, IEEE SECURITY & PRIVACY, 2005

[33] PETS, http://www.pentest-standard.org/index.php/MainPage

[34] Brad Arkin, Scott Stender, Gary Mcgraw. Software Penetration Testing. IEEE Security & Privacy, 2005

[35] OWASP, http://www.owasp.org/

[36] OSSTMM, http://www.osstmm.org/

[37] Dr. Daniel Geer, John Harthorne. Penetration Testing: A Duet. Proceedings of the 18th Annual Computer Security Applications Conference, 2002

[38] Omar H. Alhazmi, Yashwant K. Malaiya. Modeling the vulnerability discovery process. In Proceedings of the 16th IEEE International Symposium on Software Reliability Engineering (ISSRE'05). Washington, DC, USA: IEEE Computer Society, 2005

[39] Eric Rescorla. Is finding security holes a good idea? IEEE Security & Privacy, 2005

[40] Rajeev Gopalakrishna, Eugene H. Spafford. A trend analysis of vulnerabilities. Technical Report 2005-05, CERIAS, Purdue University, 2005

[41] Andy Ozment. Software security growth modeling: Examinng vulnerabilities with reliability growth models. In Dieter Gollmann, Fabio Massacci, and Artsiom Yautsiukhin, eds., Quality Of Protection: Security Measurements and Metrics. Milan, Italy: Springer, 2006

[42] Omar H. Alhazmi and Yashwant K. Malaiya. Prediction capabilities of vulnerability discovery models. In Proceedings of the IEEE Reliability and Maintainability Symposium, 2006

[43] Sung Whan Woo, Hyun Chul Joh*, Omar H. Alhazmi, Yashwant K.Malaiya. Modeling vulnerability discovery process in Apache and IIS HTTP servers. Computers & Security, Volume 30, Issue 1, 2011

[44] Viet Hung Nguyen, Fabio Massacci. An Idea of an Independent Validation of Vulnerability Discovery Models. Engineering Secure Software and Systems 4th International Symposium, ESSoS 2012, Eindhove