Presented at the First IEEE Workshop on Security Testing,
Lillehammer, Norway, April 2008
http://ieeexplore.ieee.org/document/4567022/

Learning from Software Security Testing

Inger Anne Tøndel, Martin Gilje Jaatun and Jostein Jensen
SINTEF Information and Communication Technology
Department of Software Engineering, Safety and Security
{inger.a.tondel, martin.g.jaatun, jostein.jensen}@sintef.no

Abstract

Software security testing tools and methodologies are presently abundant, and the question no longer seems to be "if to test" for security, but rather "where and when to test" and "then what?". In this paper we present a review of security testing literature, and propose a software security testing scheme that exploits an intra-organisational repository of discovered vulnerabilities that closes the loop after the testing of one application is complete, providing useful input to the next application to be tested.

1 Introduction

Few practitioners would argue against the value of performing testing of security software, or testing security mechanisms in general software products. However, security vulnerabilities have a nasty habit of cropping up in the most unlikely places! A decade ago, it might not have been unreasonable to assume that a comment field on a blog pretty much could fend for itself, securitywise. Today, we know that attacks such as SQL injection and Cross-Site Scripting turn such banalities into the proverbial barn door.

The coming of the internet age should have led developers to collectively acknowledge that all applications need to be secure, but this has not been the case. New security vulnerabilities are still being discovered in internet-accessible applications at a steady pace, and it seems as though the average developer not only suffers from a lack of interest in security [1,2], but also is incapable of learning from past mistakes [3]. In our opinion, there is thus an increased need of security also in the average software development project, and we have argued for lightweight techniques that are suitable for ensuring security in the asset identification [4], requirements elicitation [5], and design [6] phases of software development. To add another arrow to our quiver, we will in the following present an approach to software security testing that is suitable for use in all software development projects, and which facilitates learning in order to prevent developers from making the same mistakes over and over again.

The suggested testing approach complements the risk-based activities of the earlier software development phases of the SODA[1] framework [7]:

- Security requirements are identified based on high level security objectives, asset identification and high level modelling of threats towards the most important assets.
- Design guidelines, principles and patterns are, together with threat modelling, actively used to create a security architecture. A security review of the design is then performed.
- Security is taken into account when choosing programming language. Static code analysis and, if possible, a lightweight code review of the most critical parts of the application are then performed as part of the implementation phase.

Traditional software testing is mainly an exercise in Quality Assurance; "Does the application meet all the [functional] requirements [. . .] ?" [8]. If the requirements say "To get B, input A," the tester will input A, and if B is output, the test is categorised a success. Software security testing is more about testing things that *shouldn't* happen, however, and "the QA process might not be broad enough to address the true range of potentially malicious input" [9]. Furthermore, Chess and West state quite categorically that "it is almost impossible to improve software security merely by improving quality assurance" [10]. We have tried to heed this advice, and thus the security testing phase of SODA focuses on penetration testing of applications or parts of applications before deployment.

The rest of this paper is structured as follows: Section 2 presents previous work on software security testing, and in section 3 we outline the security testing cycle and identify

[1] Secure sOftware Development frAmework

where our focus lies. In section 4 we present our interpretation of the term "risk-based security testing", while section 5 sketches how a vulnerability repository may facilitate intra-organisational learning to prevent security vulnerabilities. We discuss our contribution in section 6, and offer conclusions and directions for further work in section 7.

2 Background

According to McGraw [11], it is necessary to involve two different security testing approaches: Functional and adversarial (see Table 1). As mentioned, functional security testing of security mechanisms is not a controversial strategy, but many run-of-the-mill applications will have very few (or none) of these mechanisms, rendering the *functional* security testing a relatively manageable task. Since our focus is security testing of these "average" applications, we are thus primarily interested in the *adversarial* approach. Techniques for software security testing are thoroughly covered by several books. E.g., Whittaker and Thompson [12] describe nineteen attacks that can be used to break software security, and Andrews and Whittaker [13] list 24 attacks for breaking web software; Hoglund and McGraw [14] concentrate on 49 attack patterns when describing how to exploit software; and Gallagher et al. [15] outline attack techniques that are summarised in a Security Test Cases Cheat Sheet. Several tools have also been developed to help testers, e.g. tools that focus on error handling (e.g. Holodeck[2]), monitoring of environmental interaction (e.g. Process Monitor[3] and AppSight[4] – Regmon and Filemon (now Process Monitor) and AppSight are described as promising tools by Thompson [16]) and tools for intercepting and modifying traffic between server and client (e.g. Paros[5]). For a more thorough overview of useful tools, see Gallagher et al. [15] and Andrews and Whittaker [13].

Software security testing tools and techniques have to constantly evolve to be able to detect vulnerabilities that can be utilised in new types of attacks [15]. But there are even bigger challenges:

- "One problem with almost all kinds of security testing [...] is the lack of it." [17]
- "Software development organizations tend to regard [penetration test] results as complete bug reports – thorough lists of issues to address to secure the system." [18]
- "Perhaps the most common problem with the software penetration testing process is the failure to identify lessons to be learned and propagated back into the organization." [18] "Later versions of software often contain vulnerabilities that exploit the same characteristics or conditions that were exploited by attackers in the earlier versions." [19]

Security testing is a typical last-minute activity, and the resources available scarce. Risk-based testing, defined by Amland [20] as "to focus testing and spend more time on critical functions", therefore becomes important. Some types of security vulnerabilities are more serious and/or more common than others, and statistics and rankings like OWASP Top 10[6] can be used to focus testing. Some applications, or parts of applications, can also be more likely to cause problems:

- In a presentation at the MetriCon 2.0 workshop [21], Dalci explained that the systems that tend to have the highest risk are web facing systems, large code size applications and new applications.
- Jaquith [22] suggests using complexity as an indicator for future security problems, using tools to measure cyclomatic complexity.
- Thompson et al. [23] state that error handling routines should be an important focus in testing since "many security failures occur in stressed environments, which appear in the field, but are often neglected during testing because of the difficulty to simulate these conditions".

More generally, both Gallagher et al. [15] and Wysopal et al. [8] point to threat modelling as a basis for risk-based testing. In particular, they suggest to focus on application entry points. This is also supported by McGraw [11] who suggests risk-based testing to provide "...a higher level of software security assurance than is possible with classical black box testing". Traditional black box testing usually takes an outside-in approach where the testers do not have previous knowledge of the software to be tested. Risk-based security testing, on the other hand, implies that the test process is driven by some form of risk-related input, where the risk evaluation is based on previous knowledge of the software. The risk management process gives an indication of where an attack on the newly developed software is most likely to succeed, thus testing can be focused on the most vulnerable code.

Evidence for the benefits of risk-based testing is provided by Potter and McGraw [17]. In a case study, they did both functional and risk-based testing on smartcard technologies. Via the functional tests, they found that security mechanisms often were satisfactorily implemented with respect to the defined requirements. However, when the units that previously passed the functional test were exposed to

[2] http://www.sisecure.com/holodeck/index.shtml

[3] http://technet.microsoft.com/en-us/sysinternals/bb896645.aspx

[4] http://www.identify.com/

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

[6] http://www.owasp.org/index.php/OWASP_Top_Ten_Project

No.	Approach	Why
1	Functional security testing	To determine whether security mechanisms, such as access control and cryptography settings are implemented and configured according to the requirements.
2	Adversarial security testing	To determine whether the software contains vulnerabilities by simulating an attacker's approach - based on risk-based security testing.

Table 1. Approaches to security testing [11]

a risk-based security testing approach, all of them failed! These findings emphasise the importance of structuring and prioritising the security tests based on risk. However, Potter and McGraw also express that risk-based security testing relies on expertise and experience, something that may be a problem in small organisations since most regular developers are not primarily interested in security issues.

3 The Software Security Testing Cycle

The software security testing cycle differs from the traditional testing cycle primarily in one aspect: Security testing does not have a clear-cut fulfilment criterion, and is therefore open-ended – we can go on testing 'till the cows come home, and still not be done [16]. Furthermore, there are frequently limited resources available for testing, and this results in a need to prioritise testing efforts. Thus, the first step in the cycle as illustrated in Figure 1 is to define the focus and scope of the impending security testing activity.

We acknowledge the importance of taking a risk based approach to security in all software development phases, including software security testing. This is even more important in a lightweight approach, where there is no way all security aspects can be fully addressed and tested. Our focus is on giving concrete guidelines as to how risk based security testing can be achieved in ordinary software engineering projects. In Section 4 we describe how concrete results from security techniques applied in the requirements, design and implementation phases can be used as a basis for deciding what to focus on in testing activities. These risk management activities that are tied to the specific application developed should be used together with known risk factors such as complex components, web-facing components, error handling etc. as suggested by others (see Section 2). In addition it should be taken into account where we typically have failed in the past.

As described in Section 2, various testing techniques and tools are readily available, and we therefore do not focus on the second and third step of the testing cycle, i.e. finding and fixing bugs. Instead we choose to stress that it does not stop here! For each vulnerability that is discovered by testing, special care should be taken to verify whether similar vulnerabilities exist in other parts of the same application, and root causes identified. The results from security testing must be used to improve software development, in all phases, to make sure the same mistakes are not repeated in the next version or the next project. To facilitate learning from vulnerabilities, we suggest that development organisations maintain an intra-organisation vulnerability repository, as will be further described in Section 5.

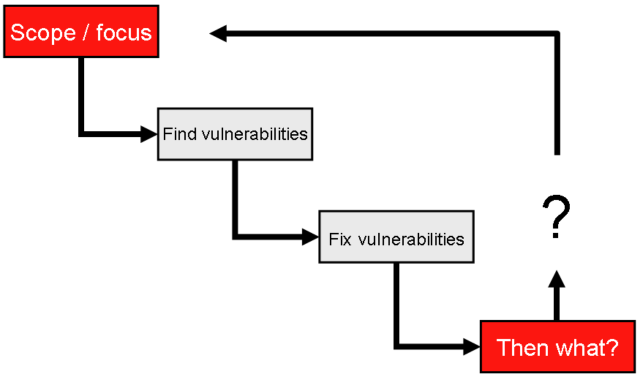

Figure 1. Software Security Testing Cycle

4 Risk-based Security Testing

Artefacts relevant to a risk-based testing will be produced in all phases of a software development lifecycle. The lightweight techniques assisting the requirements and design phases that are presented in our previous work [4–6] are no different, and they output artefacts such as lists of assets, security requirements and threat models.

The following sections describe how each of these help targeting the security testing to maximise the return on the testing effort.

4.1 Assets and Security Requirements

Assets are in many ways the very reason why we need to consider security of applications – we need security because we have something valuable that needs protection. As we see it, an important part of security requirements engineering is to produce a prioritised list of assets [4]. The most important assets are then given highest focus in threat modelling activities and when specifying security requirements [5]. In the same way, getting access to (or otherwise attacking) these important assets should be a main focus in testing activities. The security requirements will mainly give input to the functional security testing activities, which are not the main focus of this paper, but will also point to high risk areas that should be considered also for adversarial security testing.

4.2 Threat Models

Threat models will provide the most important input to the adversarial security testing. Misuse cases [24] and attack trees [25] are examples of common threat modelling techniques, and these diagrams will visualise important parts of an application's attack surface; thus pointing to the areas in the code that are most exposed to attacks. Testing must be performed on these areas to ensure that the most likely attacks are not achievable.

4.3 Static Code Analysis Results

Secure programming with static analysis is thoroughly described by Chess and West [10] and others [9, 11, 26]. Ideally, vulnerabilities reported by the static analysis tools are fixed immediately. However, it is common knowledge that these tools are not able to discover every possible vulnerability. Code segments where the returned number of vulnerabilities are above average may indicate either that the programmer has done a poor job, or that it was a particularly difficult segment to code. In both cases one should expect more vulnerabilities to lie dormant, and thus these parts should be thoroughly tested.

4.4 Choice of Language

Programming languages have different inherent security properties, and thus choice of language will influence on the testing [26]. Since e.g. C/C++ is vulnerable to buffer overflows, this is something you would want to test for; however, this would not be relevant if e.g. Java, Pascal or Ada were the programming language of choice.

It is also possible to take this idea a step further and look at which development environments that are used. Content Management Systems (CMS), for instance, are popular tools for web application development, and several of these have been reported to contain vulnerabilities. By searching through vulnerability databases (see below) for your development environment, a list of already known vulnerabilities can be found. These vulnerabilities should be tested to determine whether your application is at risk.

5 Vulnerability Repository

The vulnerability repository is our answer to the "Then what?" question in Figure 1. All vulnerabilities found by testing should be entered in the repository, as we will expand on below.

5.1 Existing Databases

Information about publicly known vulnerabilities are available at several sources like the Security Focus vulnerability database[7] and the associated mailinglist *Bugtraq*, the National Vulnerability Database[8] hosted by NIST and sponsored by the Department of Homeland Security/US-CERT, and the Open Source Vulnerability Database[9]. Common Vulnerabilities and Exposures[10] (CVE), hosted by the MITRE Corporation, provides common identifiers for vulnerabilities and exposures. For a more thorough treatment of public vulnerability repositories and mailing lists, see Ardi et al. [27].

5.2 New Vulnerabilities

Knowledge about publicly known vulnerabilities is important and can be utilised by software development organisations. Different organisations have different characteristics, however; culture, knowledge-level or special properties of the applications being developed can be the reason why some vulnerabilities are more common than others. Knowledge of what are the most commonly introduced vulnerabilities can be used to focus testing, but also to improve the software development process.

Thompson [28] has argued that "Bugs are corporate assets that should be treasured and studied." By analysing vulnerabilities it is possible to understand what the organisation is doing wrong, and compensate through the use of secure development techniques. To better enable learning from own mistakes, we suggest organising information about all vulnerabilities found in an intra-organisational vulnerability repository, as illustrated in Figure 2. This

[7] http://www.securityfocus.com/archive/1

[8] http://nvd.nist.gov/

[9] http://osvdb.org/

[10] http://cve.mitre.org/

repository should include vulnerabilities found during security testing, but also design flaws found during design review, coding mistakes found by static analysis tools or code reviews (if applied), and vulnerabilities found after deployment. It is important to learn from vulnerabilities regardless of when they are detected.

5.3 Goals

The goals we hope to achieve by introducing the vulnerability repository include:

- **Improve the ability to produce secure software:** By using the vulnerability repository actively to guide the security development process in the organisation, it should be possible to reduce the number of vulnerabilities in software, especially vulnerabilities that have traditionally been common or have been given focus because of the high risk involved.

- **More cost-effective handling of vulnerabilities:** Focus on common vulnerabilities should result in these vulnerabilities being avoided alltogether or detected at earlier stages where the cost of fixing vulnerabilities is at a minimum [29].

- **Measure progress:** The vulnerability repository can be used to measure progress, i.e. if vulnerabilities are detected sooner in the development process, or if the number of vulnerabilities are reduced. Such measurements can be a motivation factor. Note however that too much focus on reduced number of vulnerabilities can result in less effort when it comes to finding vulnerabilities, and thereby reduced quality of testing. Finding many vulnerabilities is a good thing and should also be appreciated.

5.4 Using the Repository

Information from the vulnerability repository should be utilised at all stages of the development process in order to avoid or detect the vulnerabilities as early as possible:

- **Requirements engineering:** Vulnerabilities that are common and potentially high risk can be used as input to general software development policies that apply to all applications developed by the organisation. Example: All applications shall have proper input handling. These general policies will then be used as a basis for security requirements together with customer requirements, asset identification and threat modelling activities [5].

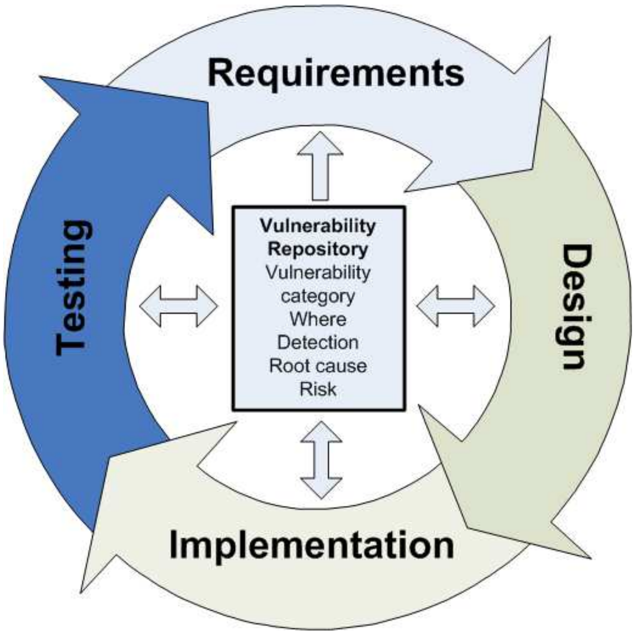

Figure 2. Vulnerability repository

- **Design:** Knowledge of common vulnerabilities can guide software designers to make more secure design choices, like choosing appropriate security design guidelines, principles and patterns. As an example: Statistical evidence that an organisation's software is susceptible to SQL injection attacks may be used as an argument to include the Intercepting Validator security pattern defined by Steel et al. [30]. A design review should also profitably be based on past experiences.

- **Implementation:** The choice of language and the use of frameworks can influence which types of implementation vulnerabilities that are common. If large groups of vulnerabilities can be removed by changing e.g. programming language this should be taken into account when making such decisions. Knowledge of common vulnerabilities can also be used to focus code reviews, if the organisation is performing such reviews. Finally, knowledge of common implementation errors can be used to tune the rule-sets used in the static code analysis tools.

- **Testing:** Information on where we typically have failed in the past should be used as input when prioritising testing efforts. Concrete vulnerabilities as well as statistics on which vulnerability categories are most common should be utilised. This type of information is easily obtained from the suggested vulnerability repository.

If lack of knowledge and training is identified as a cause for important groups of vulnerabilities, this information should be used to focus training initiatives. Concrete vulnerabilities should then be utilised for motivation.

5.5 What to Record

To limit the work related to registering and follow-up of vulnerabilities, we suggest to strive towards only registering the information we intend to use. We suggest the following information as a minimum:

- Date – to be able to measure progress over time
- Vulnerability category – to be able to know what areas to focus on in improvement activities
- Where – to be able to know what portions of the code and what aspects of the application to focus on when improving security development techniques
- In which phase and with which technique it was detected – to make sure the vulnerability is tested for in the future, and also to be able to see if the same type of issue is detected earlier in future projects
- Root cause – to use as input to improve the security development process
- Risk – to prioritise what to focus on, but also to see if the risk posed by the vulnerabilities detected decreases over time
- Countermeasure – to learn how this type of vulnerability can be prevented or avoided, e.g. by referring to a relevant security pattern.

In addition, it may be of interest to record who introduced the vulnerability to allow each developer to learn directly from their own mistakes. However, it is important to consider possible side effects, such as employees feeling they are being publicly embarrassed and become uncomfortable with their working environment.

For the registration of vulnerabilities, we suggest to utilise predefined categories to ease aggregation and searchability of information, and use free text for details. Regarding vulnerability category, it is advantageous to use existing vulnerability taxonomies like the 7+1 kingdoms defined by Tsipenyuk et al. [31], and detail these if necessary. For describing risk it is possible to utilise the Common Vulnerability Scoring System [32]. By representing the vulnerabilities in a standard way it will be easier to share vulnerability information in an anonymised and generalised form, so that they can be integrated in a public or federated repository, as suggested by Ardi et al. [27].

6 Discussion

In this paper we have argued for the introduction of an intra-organisational vulnerability repository, and we have explained how this can be used to improve the security awareness throughout all phases of a development lifecycle. Our key motivation has been increased focus on risk-based security testing in all software development projects, and improvements in the way development teams learn from their mistakes. The vulnerability repository is currently a sketch-board idea that need to mature and be tested to have real knowledge about its usefulness. In the following we will however discuss possible strengths and weeknesses of using such a reposiroty in combination with risk based testing.

Large software development organisations frequently have separate development teams and security teams because of the complexity of security work. Other organisations may even hire external security experts to evaluate their software and do the security testing. [15] In such situations, the importance of a vulnerability repository is highlighted: If knowledge about vulnerabilities discovered by the security testers is not passed on to the actual developers who introduced the vulnerabilities, they will keep making the same mistakes over and over again.

Establishing databases with information on reported security vulnerabilities is nothing new, e.g. Howard and Lipner [33] describes bug tracking databases as important assets in the Secure Development Lifecycle (SDL) process. However, their main use of such databases is to keep lists of errors to fix before a software release, or lists of vulnerabilities reported by the public (for which security patches should be produced). Our approach has a wider scope, where the focus is to produce more secure software - also in later projects.

One drawback of our approach may be that it in many ways is (too) closely tied to traditional development methods, and consequently would be less suited to an agile development context. Most of the artefacts we recommend as input to the risk-based testing activity are never created in agile methods such as eXtreme Programming (XP). On the other hand, the XP tenet of "write the test first" could have interesting implications if re-written "write the *security* test first." Whether this is in fact possible, remains an open question.

As computer security professionals, the importance of increased software security is abundantly clear to us. However, we also realise that the average developer might be less than thrilled when presented with yet another reporting scheme that requires constant monitoring and maintenance; this can easily be perceived as an extra burden that does not contribute appreciably to the implicit goal of producing the most functionality in the least possible time.

Thus, for a software security testing scheme to be successful in the average software development project, it has to be easy to use and require an acceptable amount of resources. Still, it must be able to detect the most severe vulnerabilities. More practical experience is needed in order to confirm whether this is achieved with our suggested approach, though we believe the practical recommendations will improve security testing in most projects and enable teams and organisations as a whole to learn from past mistakes. To be able to utilise the full potential of the vulnerability repository it is however necessary to integrate the use of the repository into working routines, and even better, to integrate the repository with development tools so that relevant information is available when needed [27].

Our work is mainly targeted towards developers and testers who are not security experts. By following our lightweight approach they will be provided with sufficient basis to organise and prioritise security testing. We do not claim that we have presented the perfect solution, however, we do argue that a little security focus goes a long way. We propose to base the risk-based testing scheme on artefacts that (in our opinion) need to be developed anyway, thus minimising the extra effort required to identify test targets.

7 Conclusion and further work

We have presented a lightweight software security testing scheme where the focus is on utilising the limited resources available in the best possible way. Concrete guidelines are offered on how to take a risk based view on testing, assuring that the testing efforts are focused on the most critical parts of applications. An intra-organisational vulnerability repository is suggested in order to enable learning from vulnerabilities at all stages of software development. In this way, organisations do not have to use resources on fixing the same problems over and over again in different software versions and projects.

More work is needed on integrating the vulnerability repository into the overall software development process, and especially on tool support. This includes utilising information from the repository together with the other risk-based techniques to generate concrete recommendations on what to focus on in security testing. An important steps towards achieving this is to determine how to categorise or otherwise represent vulnerabiities. More work is also needed on how to weigh different factors, like failures from past project vs. important assets, to be able to provide more concrete recommendations. The possible advantages of connecting intra-organisational vulnerability repositories with external repositories should also be more thoroughly addressed, since this is likely to influence what vulnerability information to register, and may increase the advantages of integrating the repository with common development tools. However, such integration may also pose new challenges with respect to generalisation and anonymisation, since un-anonymised vulnerability information may be considered highly confidential.

Acknowledgments

The authors wish to thank Per Håkon Meland for inspiring us to write this paper, and for providing useful comments. We also wish to thank the anonymous reviewers.

References

[1] P. Coffee. (2006) Security Onus Is on Developers. eWeek. [Online]. Available: http://www.eweek.com/article2/0,1895,1972593,00.asp

[2] H. Mouratidis, P. Giorgini, and G. Manson, "When security meets software engineering: a case of modelling secure information systems," *Information Systems*, vol. 30, no. 8, pp. 609–629, 2005.

[3] C. Cowan, P. Wagle, C. Pu, S. Beattie, and J. Walpole, "Buffer overflows: Attacks and defenses for the vulnerability of the decade," in *Proceedings of DARPA Information Survivability Conference and Expo (DISCEX)*, Hilton Head Island SC, January 2000.

[4] M. G. Jaatun and I. A. Tøndel, "Covering your assets in software engineering," in *The Third International Conference on Availability, Reliability and Security (ARES 2008)*, Barcelona, Spain, 2008, pp. 1172–1179.

[5] I. A. Tøndel, M. G. Jaatun, and P. H. Meland, "Security Requirements for the Rest of Us: A Survey," *IEEE Software*, vol. 25, no. 1, 2008.

[6] P. H. Meland and J. Jensen, "Secure software design in practice," in *The Third International Conference on Availability, Reliability and Security (ARES 2008)*, Barcelona, Spain, 2008, pp. 1164–1171.

[7] (2007) SODA – a Security-Oriented Software Development Framework. SINTEF ICT. [Online]. Available: http://www.sintef.no/soda

[8] C. Wysopal, L. Nelson, D. D. Zovi, and E. Dustin, *The Art of Software Security Testing: Identifying Software Security Flaws*. Symantec Press, 2006.

[9] M. Dowd, J. McDonald, and J. Schuh, *The Art of Software Security Assessment*. Addison-Wesley, 2007.

[10] B. Chess and J. West, *Secure Programming with Static Analysis*. Addison-Wesley, 2007.

[11] G. McGraw, *Software Security - Building Security In*. Addison-Wesley, 2006.

[12] J. A. Whittaker and H. H. Thompson, *How to Break Software Security*. Addison-Wesley, 2003.

[13] M. Andrews and J. A. Whittaker, *How to Break Web Software*. Addison-Wesley, 2006.

[14] G. Hoglund and G. McGraw, *Exploiting Software: How to break code*. Addison-Wesley, 2004.

[15] T. Gallagher, L. Landauer, and B. Jeffries, *Hunting Security Bugs*. Microsoft Press, 2006.

[16] H. Thompson, "Why security testing is hard," *Security & Privacy Magazine, IEEE*, vol. 1, no. 4, pp. 83–86, July-Aug. 2003.

[17] B. Potter and G. McGraw, "Software security testing," *Security & Privacy Magazine, IEEE*, vol. 2, no. 5, pp. 81–85, Sept.-Oct. 2004.

[18] B. Arkin, S. Stender, and G. McGraw, "Software penetration testing," *Security & Privacy Magazine, IEEE*, vol. 3, no. 1, pp. 84–87, Jan.-Feb. 2005.

[19] K. Jiwnani and M. Zelkowitz, "Maintaining software with a security perspective," in *Proceedings of the International Conference on Software Maintenance (ICSM'02)*, Montréal, Canada, 2002.

[20] S. Amland, "Risk based testing: Risk analysis fundamentals and metrics for software testing including a financial application case study," *The Journal of Systems and Software*, vol. 53, pp. 287–295, 2000.

[21] D. Geer. (2007) MetriCon 2.0 Digest. [Online]. Available: http://www.securitymetrics.org/content/attach/Metricon2.0/metricon2.digest.PDF

[22] A. Jaquith, *Security Metrics: Replacing Fear, Uncertainty, and Doubt*. Addison-Wesley, 2007.

[23] H. H. Thompson, J. A. Whittaker, and F. E. Mottay, "Software security vulnerability testing in hostile environments," in *SAC '02: Proceedings of the 2002 ACM symposium on Applied computing*. Madrid, Spain: ACM, 2002, pp. 260–264.

[24] G. Sindre and A. L. Opdahl, "Eliciting security requirements with misuse cases," *Requirements Engineering*, vol. 10, no. 1, pp. 34–44, 2005.

[25] B. Schneier, "Attack Trees - Modeling security threats," *Dr. Dobb's Journal*, July 2001. [Online]. Available: http://www.ddj.com/184411129

[26] K. M. Goertzel, T. Winograd, H. L. McKinley, L. Oh, M. Colon, T. McGibbon, E. Fedchak, and R. Vienneau, "Software Security Assurance," Information Assurance Technology Analysis Center and Data (IATAC) and Analysis Center for Software (DACS), Tech. Rep., 2007.

[27] S. Ardi, D. Byers, P. H. Meland, I. A. Tøndel, and N. Shahmehri, "How can the developer benefit from security modeling?" in *The Second International Conference on Availability, Reliability and Security (ARES 2008)*, Vienna, Austria, 2007.

[28] H. Thompson, "Application penetration testing," *Security & Privacy Magazine, IEEE*, vol. 3, no. 1, pp. 66–69, Jan.-Feb. 2005.

[29] K. Hoo, A. Saudbury, and A. Jaquith, "Tangible ROI through Secure Software Engineering," *Secure Business Quarterly*, vol. 1, pp. 1–3, 2001.

[30] C. Steel, R. Nagappan, and R. Lai, *Core Security Patterns: Best Practices and Strategies for J2EE(TM), Web Services, and Identity Management*. Prentice Hall, 2005.

[31] K. Tsipenyuk, B. Chess, and G. McGraw, "Seven pernicious kingdoms: a taxonomy of software security errors," *Security & Privacy Magazine, IEEE*, vol. 3, no. 6, pp. 81–84, Nov.-Dec. 2005.

[32] P. Mell, K. Scarfone, and S. Romanosky, "Common vulnerability scoring system," *Security & Privacy Magazine, IEEE*, vol. 4, no. 6, pp. 85–89, Nov.-Dec. 2006.

[33] M. Howard and S. Lipner, *The Security Development Lifecycle*. Microsoft Press, 2006.