Proceedings of the 10th Colloquium for Information Systems Security Education
University of Maryland, University College
Adelphi, MD June 5-8, 2006

Teaching the Secure Development Lifecycle: Challenges and Experiences

Rose Shumba, *Member IEEE*, James Walden, *Member IEEE*, Stephanie Ludi, Carol Taylor, *Member IEEE*, and Andy Ju An Wang

Abstract – ***A large portion of security vulnerabilities result from mistakes in the design or code of software systems. To address this problem, secure development lifecycle practices have been introduced into the software engineering curriculum at five different universities. Each phase of the software development lifecycle has been modified in at least one university to incorporate security. This paper provides a survey of practices involved in the secure development lifecycle and describes how these practices can be introduced into the software engineering curriculum. Each contributor discusses his or her experiences and challenges while integrating security into one phase of the software development process.***

Index terms – Software assurance education, software security, secure development lifecycle, secure programming.

I. Introduction

Security flaws are endemic to computer software; CERT reported over 5000 software vulnerabilities last year. Exploitation of these flaws by attackers and malicious software cost US industry and government over ten billion dollars per year and potential attacks on critical infrastructure remain a serious concern. In large part, these vulnerabilities are the result of software vendors employing a "penetrate and patch" approach to security problems, addressing security issues only after external entities discover flaws in their software.

Typical software development life-cycle models do not focus on creating secure systems, and fall short when the goal is to develop systems with a high degree of assurance [1]. When security issues are addressed, they are often relegated to a separate thread of project activity, with the result that security is treated as an add-on property. This isolation of security considerations from primary system-development tasks results in an unfortunate separation of concerns [2].

The state of industrial secure development practice has begun to change. Microsoft began their push for Trustworthy Computing in 2002[3] and published their Trustworthy Computing Secure Development Lifecycle in March 2005[4]. In 2004, the National Cyber Security Partnership (NCSP) task force, a group of security technology experts, academics, and business and government officials, released a 123-page report[5] to the U.S. Department of Homeland Security. The report issued preliminary recommendations for improving software security by addressing security throughout all phases of the software development lifecycle. Among the key recommendations:

- Awareness & Education: Improving the education of current and future software developers.
- Process Improvement: Adopting software development practices that can measurably reduce software specification, design and implementation defects.
- Redesign of Flawed Systems: Encouraging software producers to recognize systems with unacceptable architectures and designs and re-architect and redesign them with proper characteristics for security, using quality software development processes.
- Security Best Practices: Interleaving security best practices throughout the software design process.

The Department of Homeland Security's "Build Security In (BSI)" Software Assurance Initiative [6] then emanated from the NCSP report in 2005. The BSI initiative marked a new government focus on secure software development, including coordinating the development of a Software Assurance Common Body of Knowledge. The initiative also includes a BSI content catalog available on the U.S. Computer Emergency Readiness Team Web site at <http://BuildSecurityIn.us-cert.gov>. The catalog is based on the principle that software security is fundamentally a software engineering problem and must be addressed in a systematic way throughout the software development life cycle[7].

Security-enhanced processes and technologies are necessary to build trust into software acquired and used by the broad spectrum of business and mission operations, including everything from process control systems to commercial application products that support and enable

Rose Shumba, Indiana University of PA
Carol Taylor, University of Idaho
Stephanie Ludi, Rochester Institute of Technology
James Walden, Northern Kentucky University
Andy Ju An Wang, Southern Polytechnic State University

ISBN 1-933510-98-6/$15.00

them. Security must be given the same level of consideration as other software requirements in order to create software that can withstand failures and compromises.

Few university software engineering courses or textbooks incorporate secure software engineering practices. The few books on the topic are relatively new and only document collections of best practices. Most courses and textbooks focus on secure coding [3, 9-11], which is only one phase in the development process. Although these resources are invaluable, they can lead developers to conclude that the application is secure once well known flaws like buffer overflows or format strings are handled. However, two books focused on professional practice have been announced that look like they will cover the entire secure software development process[12-13].

Training of present and future developers, programmers, software architects, and project managers should put security at the foundation of the development process. However, incorporating the secure development lifecycle into the software engineering curriculum presents a considerable challenge due to the time limitations of existing software engineering classes and the immaturity of secure software engineering methodologies and tools.

The secure software development process follows an iterative process of requirements analysis, design, secure implementation, and security testing. A threat model, which describes the threats the application needs to mitigate, drives the software assurance process through each stage of the life cycle. During requirements gathering, students document threat models with misuse cases to ensure that security is given the same importance as functional requirements documented with use cases.

In the design phase, the threat model is expanded on by adding data flow diagrams. Secure design principles guide the design process. Security mechanisms like access control and cryptography are introduced to counter the threats documented in the threat model. Implementation phase activities include training developers to avoid common coding flaws and reviewing code for vulnerabilities. Security testing expands on regular test activities by simulating various attack scenarios to test the system's resistance to real life attacks.

This paper is divided into four major sections, one for each major phase of the secure development lifecycle: requirements, design, implementation, and testing. Each section is written by an author who has experience integrating secure practices into a class covering that phase of development. Each author discusses the challenges they faced incorporating security into that phase and their experiences teaching it.

We provide secure development practices that can be incorporated into a wide variety of software engineering courses, ranging from a single course in software engineering or computer security to specific courses focused on requirements, analysis and design, secure coding, or verification and validation. The challenges described in each section alert instructors to potential difficulties teaching the secure development practices described, while the experiences that follow offer a guide to resolving those issues.

II. REQUIREMENTS

As part of a college-wide graduate degree in Computer Security and Information Assurance (CSIA), the author taught an inaugural offering of a graduate course entitled Secure Software Engineering: Analysis & Design during the Winter 2005 quarter. The course is required for graduate students in the MS CSIA program, and a small group of upper-division undergraduates in the Software Engineering program were allowed to enroll in the course as well.

The course is designed to be highly interactive, and to include problem-based and exploratory learning. Course readings consisted of popular texts and academic papers, enabling a blend of recent research and techniques that are applicable to a variety of projects [3, 14]. While early sessions of the course were dominated by lectures, the majority of the course meetings were driven by questions and exercises proposed at the beginning of the course meeting. The questions and exercises reflected the assigned reading and challenged the students to think critically. While the students worked in small teams, the instructor met with the students to clarify material, ascertain understanding and to ask follow-up questions. At the end of the course meeting, the students would share their responses and discuss their findings. To extend the collaboration and to encourage the graduate students to explore topics that interest them, student teams posted their material in the online forum. An ongoing challenge was to offer a challenging, useful, and interesting course for software engineering undergraduates who had a solid foundation in all aspects of software engineering (except security) while not intimidating the majority of graduate students who have only had an introduction to software engineering. Only one of the ten graduate students was a graduate of the software engineering program.

The main vehicle of the course was a team project. Students worked on a 9-week project in teams of 4-5 students. Each student team was free to propose a project to analyze, reverse engineering the requirements and design from an existing code base. Three different open

source projects were used by the students: Gallery2, Scatterchat, and Wildfire Server. While some similarities existed between the project domains, the different technologies and designs provided for an interesting experience for all. A side effect that also reflects the reality of software development was the lack of documentation. The main project deliverables consisted of:

1. The development of the project SRS, including misuse cases and risk assessment
2. A threat model for a subset of the system
3. The development of the project architecture document
4. An architecture audit report for the system
5. A presentation that encompassed both requirement and design components

Usually an iterative process is used in software engineering project; however, a waterfall process was used in order to accommodate the time needed to assess the system at a sufficiently deep level for the course. Discussion of the application of security practices in other process models was included in the course, including that of agile development approaches and Microsoft's Trustworthy Computing Security Development Lifecycle[4].

The low-level topics presented in the course enabled students to relate the new material to the concepts associated with traditional software engineering and to apply the new concepts to the course project. The main concepts discussed were the role of risk, threat modeling, misuse cases, and analysis patterns and antipatterns. Misuse cases require students to expand their thinking about the potential (mis)use of the systems that are being specified [15-17]. The technique is similar to use cases, so students accept the technique readily. The application of misuse cases to a real project offered students the opportunity to apply the technique to a nontrivial system. A template was provided based on their reading. The development of the project's SRS provided benchmarks for subsequent project activities and analysis. The notion that security takes various forms was always included in the course, including the idea that systems need to survive attacks. The project deliverables reflected this notion as well.

Threat modeling, specifically through the use of attack trees, was presented at both the requirements and design phases of the project. This dual examination allowed for the reflection of the nature of attack and analysis at different phases of development. The use of the graphical notation of attack trees was a useful exercise, and offered the students the means to extend the use of risk assessment in their projects [18]. An in-class activity, where students created attack trees for their projects extended the value of the risk assessment discussion. Subsequent system audit work used the attack trees as the basis to direct the audit of the security requirements and the architecture.

To complement the discussion of design patterns later in the course, patterns and antipatterns in the area of requirements analysis were included in the course. The discussion of analysis patterns and formal analysis was based upon academic readings selected to show the breadth of work being conducted in the area of security in requirements engineering [19-20]. For those students who had limited experience with patterns, an introduction to patterns with an analysis perspective eased the learning curve for subsequent work in design.

The key to including requirements engineering in a secure software engineering course is to provide students with concrete techniques and knowledge that can be applied to real projects. During a mid-quarter feedback session, some students expressed frustration with the academic readings (as opposed to the use of a textbook). Students were overwhelmingly pleased with techniques that they could apply to a project. The instructor will carefully evaluate readings for the next offering of the course.

III. DESIGN

As part of a single semester software engineering class, the author taught secure design practices, including threat modeling and secure design principles. Secure design expands on the threat model begun in the requirements phase. The initial threat model documents the assets that may be targeted by attackers, the entry points that may be used to gain access to those assets, and the threats presented by potential attackers. The task of the design phase is to document how input traverses the system from entry points to assets and to design security mechanisms to mitigate threats to those assets.

Designers construct data flow diagrams (DFDs) [21] to document the movement of data through the system. They start by constructing a context diagram that illustrates the system's interactions with external entities, such as users, databases, and external authentication and auditing systems. The system is then decomposed into more detailed DFDs in a hierarchical system until no privilege boundaries exist between nodes in the diagram. Privilege boundaries indicate that nodes on each side of the boundary have different trust levels associated with them.

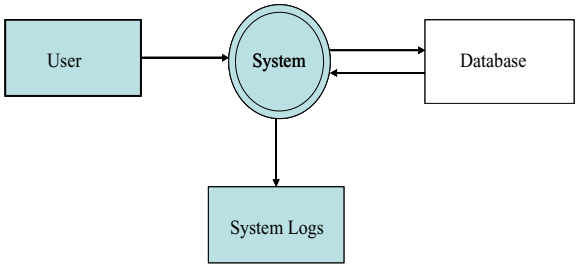

FIGURE 1
CONTEXT DATA FLOW DIAGRAM

When using an iterative software development process, student teams can be given an initial set of threats to address in their application, including assets targeted by the attacker and entry points that will be used. These threats will be only ones involved in evaluating the security of the student's application in the first iteration. While students work on the initial iteration, instructors explain general patterns of attack to students and common attacker goals and students are responsible for expanding their threat model in the second and subsequent iterations to account for new threats inspired by these attack patterns. The threat model may also be updated when new threats are discovered during testing.

While the threat model documents threats to an application and where to defend against them, it doesn't offer guidance as to how to defend against those threats. Secure design principles offer guidelines to help designers counter threats and avoid common security flaws in their application. Saltzer and Schroeder [22] provide a list of eight secure design principles:

- Economy of Mechanism
- Fail-safe defaults
- Complete Mediation
- Open Design
- Separation of Privilege
- Least Privilege
- Least Common Mechanism
- Psychological Acceptability

Some design principles like the principle of least privilege are easily understood by students because of their familiarity with concepts like file access control lists (ACLs.) However, some easily understood principles aren't often followed. It's easy to "fix" an access denied error by setting ACLs so that resources are readable and writable by everyone or by giving program excessive privileges. We have observed students applying this type of solution in a variety of classes.

Other principles are not readily understood by most students, as commonly used software does not apply principles such as that of complete mediation. Most widely used filesystems do not follow the principle of complete mediation, as ACLs are only checked when a file is opened. Subsequent reads and writes do not check the ACL, as the principle requires. If a user changes the ACL after the file is opened to forbid access, there is no effect on processes that opened the file prior to the change.

The primary challenge in teaching secure design is that students lack the design experience to apply these principles in their initial designs. In particular, they become confused when two principles conflict. For example, software designers are often confronted with the question of what product features to enable on installation. Enabling certain features, such as open network ports or the ability to execute macros, changes the security configuration of the system. The principle of fail-safe defaults guides designers to enable only features necessary for the system's operation, leaving all optional features disabled, while the principle of psychological acceptability points users to increase ease of use by enabling all features that users might wish to use.

We can give students the necessary experience to make such design decisions by showing them case studies of secure system design [23-24] and by using an iterative development process. Iterative processes give students multiple chances to make and evaluate design tradeoffs. After testing and using their system, students can discover a set of features to enable on installation that balances the principles of fail-safe defaults and psychological acceptability.

IV. IMPLEMENTATION

In the summer of 2003, with funding from an NSA capacity building grant, we started work on incorporating security issues into two software engineering courses: Software Engineering Concepts (COSC319) and the Software Practice (COSC320). COSC319 is a prerequisite for COSC320. Security topics have only been integrated into the COSC320 course so far.

COSC320 is a team project course with an average semester enrollment of 16 students. Two teams of seven or eight students each build a web-application development project for a client. Typical web-based projects assigned over the past few semesters include a health management system, an energy monitoring system, and a help desk application. The client provides a project requirements document. Our main client has been the Application Development Office of the Technology Services Center at IUP.

The development process used for the project shares both iterative and incremental characteristics. The main functionality increments of the system are identified, then

development is organized into a series of fast increments for specification, development, and certification.

During the implementation phase, student teams choose and justify choice of development language, code the increments, then conduct code review sessions for each major developed increment. Class time is spent discussing common vulnerabilities, including environment variables, buffer overflows, format strings and programmers backdoors.

Student teams are free to choose a language for project development. The choice of the programming language strongly impacts the development process for creating a secure software project, as different languages expose the developer to different security risks. For example, buffer overflows are a common type of security vulnerability in C and C++, but aren't a major problem in languages that provide automatic garbage collection like Java or Python. The choice of language also influences which security libraries and tools are available or important to the development process. However, it is important to stress to students that use of a particular language does not guarantee security. While use of a language with garbage collection protects developers against most buffer overflows, most languages don't prevent developers from writing common web application vulnerabilities, such as SQL injection or cross-site scripting. Java and PHP are usually chosen for the project.

During implementation, students refer back to the threat model and misuse cases developed during analysis and design to ensure that particular attention is given to the coding of modules that mitigate high-priority threats.

After implementing a major component, students carry out code reviews. Code reviews are essential to ensure that code is checked for security flaws before integration, ensuring that identified threats are blocked and mitigated. A venue outside the usual classroom setting is chosen; a departmental board room is often used. Refreshments (coffee and tea and water) are provided. The duration of code review sessions varies depending on the complexity of increment, but typically is about an hour. In preparation for a code review session, four formal roles are assigned: the Moderator, Reader, Recorder, and Author. The Moderator, who must be very competent technically, leads the inspection process. He or she paces the meeting, coaches other team members, deals with scheduling a meeting place and disseminating materials before the meeting, and follows up on any rework. The Reader takes the team through the code by paraphrasing its operation. A Recorder notes each error on a standard form, freeing other team members to focus on thinking deeply about the code. The Author's role is to understand the errors found and to illuminate unclear segments of code. Reading competence is most obviously relevant to program maintenance, for which the programmer must gain sufficient understanding of the code to design modifications that extend, adapt, or correct it, while retaining its logical, functional, and stylistic integrity. Before a code review session, students are expected to work individually in trying to understand the program. They document the meaning of every vital statement in the code. Understanding and reading a program is an essential skill for all programmers. Effective code verification and code reviews require students to understand and analyze the code under scrutiny. A set of security developer guidelines and checklists [32] are used in the code review. Checklists help the students with the list of items to be checked or remembered. Students are also given language-specific guidelines [33] for secure programming practices.

At the end of the code review session, students produce a report documenting potential flaws, mitigation techniques, challenges faced, and solutions discovered. One observation from reports produced so far is that students are more concerned about functional than security requirements, and therefore spend more time reviewing functional requirements. It is vital that the instructor serve as the Moderator for the first code review.

Besides providing a crucial step in the process of removing security vulnerabilities from software during the development process, code reviews help develop students' communication and program comprehension skills. During the final project presentation done for the client, we observe substantial improvement in these areas. Students appreciate code reviews as a good way to examine source code and to detect and remove potential security vulnerabilities. Although students rate code reviews highly, reviews take a substantial amount of time to organize and conduct.

Another author of this paper has added static analysis tools to the code review process. Even when using checklists, developers will miss some security flaws in the code being reviewed, so static analysis tools are used to help reviewers find all of the problems with their code. A variety of open source static analysis tools are freely available, while some commercial tools offer affordable academic licenses.

However, static analysis tools cannot replace knowledgeable reviewers. These tools produce both false positive and false negative results. The first generation of such tools, including ITS4, RATS, and flawfinder, perform simple text searches of source code for strings representing bad code, such as gets(), and produce so many false positives as to be almost unusable. Newer tools, like splint, parse the source code, and produce better results, though they still produce many false

positives and are difficult to use. Commercial tools like Fortify's Source Code Analyzer produce many fewer false positives and offer a user interface that makes it possible to easily track data flow from the user to the vulnerable code, so we recommend using such tools if they are available.

V. TESTING

Software can be correct without being secure. The symptoms of security vulnerabilities are very different from those of traditional bugs [27]. Various testing methods have been discussed in software engineering literature including black-box testing, white-box testing, equivalence partitioning, structural testing, path testing, and integration testing. Unfortunately none of these testing methods work well for software security testing. The key reason for this is that security requirements are inherently difficult to formalize. Security breaches and malicious attacks are unpredictable and difficult to be tested through conventional test cases.

Traditional testing methods assume a perfect world for software to run in concerns of security: the users of the software are perfect, who never attempt to misuse an application for revenge their former employers or for illicit personal gain; the environment of the software is perfect, it never interacts with the software with hostile return values; the API or library functions are perfect, they never refuse the request from the software under testing and they always deliver the correct values in the right temporal order [27]. Unfortunately these assumptions are largely untrue in the real world.

As an example, buffer overflow have been exploited by roughly 50% of security attacks. Unfortunately buffer overflow vulnerability can hardly be tested with the current testing methods targeting program correctness. Widely used operating systems like Solaris, Linux, and Windows have been thoroughly tested using traditional software testing methods before release. However, security breaches and new vulnerabilities in these operating systems have been reported every month. Software security vulnerabilities have been identified in several products used to increase computer security or manage passwords, including Zone Lab's ZoneAlarm personal firewall, McAfee Security ePolicy Orchestrator, and the open-source Password Safe [28].

As expressed in [27], traditional software bugs can be found by looking for behaviors that do not work as specified. Security bugs, however, are often found by looking at those additional behaviors, side effects, and the implications of interactions between the software and its environment. These additional behaviors are not specified in software requirements specification, therefore, they must be verified using non-traditional testing method – security testing. The security testing tasks include penetration and destructive tests that are different from functional testing tasks currently covered in software engineering textbooks.

At the School of Computing and Software Engineering at Southern Polytechnic State University, we have been trying to incorporate security issues in two software engineering courses: SWE 3683/6823, Embedded Systems Analysis and Design, and SWE 3843/6843, Embedded Systems Construction and Testing. Both courses were designed for senior undergraduates and first year graduate students with an introduction to software engineering as the prerequisite. Many embedded systems perform safety-critical missions such as auto-pilot or weapon control; others are business-critical applications within the data and telecom sector. For SWE 3683/6823, we emphasize the importance of security in embedded systems and teach students how to create a secure software product using a secure development lifecycle. For SWE 3843/6843, we focus on security testing in addition to traditional functional testing. We adopted a Relatively Complete Coverage (RCC) principle for security testing, which is an adapted version of the complete coverage (CC) principle for functional testing presented in [29]. An automatic security testing tool is presented then followed by a brief discussion on the challenges in teaching security testing.

Wang [31] reported how to apply the RCC principle to security testing for different software applications. Students conducted functional testing first, then performed security testing on assigned programs. They found that these two steps of testing helped them understand the importance of security testing and gain insights on developing secure software. The RCC principle discussed provides only a guideline for systematic security testing. The application of RCC would be difficult without tool support. Fortunately [27] provided a security testing tool called Holodeck that helps to conduct security testing. As shown in Figure 2 below, this tool provides a number of run-time constraints checking wizards including networking, memory, and disk.

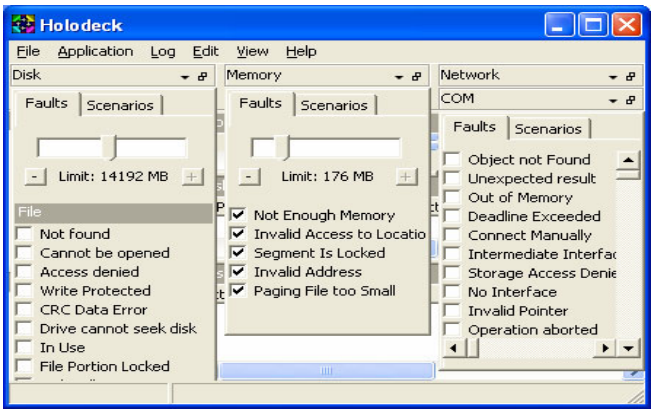

FIGURE 2
A WINDOW CAPTURE OF HOLODECK

There are two challenges in teaching security testing for embedded software. First, students tend to think that embedded software is just a special kind of software and security is just one of the non-functional requirements along with such aspects as performance and reliability. Second, students tend to think thorough testing would be the best way to improve the software quality and satisfy the security requirements. In addition to using different testing tools with RCC approach, we are investigating how to develop secure and trustworthy embedded software following the Microsoft's Security Development Lifecycle [4].

VI. CONCLUSION

This paper described the integration of security into the software development lifecycle. Each contributor outlined secure development practices incorporated into a specific stage of the lifecycle in a single course. Secure requirements analysis, design, implementation, and testing practices were incorporated into a variety of courses, including a single semester software engineering course, a secure analysis and design course, a secure coding course, and two secure embedded systems courses.

Teaching the secure development lifecycle presents a considerable challenge due to time limitations, lack of textbooks, and the immaturity of secure development methodologies and tools. However, adapting robust coding and testing processes to secure development requires little additional class time, and much can be accomplished without extensive alterations of the software engineering curriculum by selecting examples and projects that have security relevance.

Course evaluations indicated increased awareness and knowledge of software security. Some of the authors are planning quantitative measures of student learning for future courses. Comparative evaluations of student ability to identify security threats, design successes and mistakes, and coding errors given at the beginning and end of a software engineering course would offer insight into how well students are learning secure development practices.

VII. REFERENCES

1. Marmor-Squires, A.B., and Rougeau, P.A. Issues in Process Models and Integrated Environments for Trusted Systems Development. Proc. of the 11th National Computer Security Conference. Fort George G. Meade, MD, Oct. 17-20, 1988: 109-113.
2. Mead, N.R., et al. "Managing Software Development for Survivable Systems." Annals of Software Engineering 2 (2001): 45-78.
3. Howard, M., LeBlanc, D. *Writing Secure Code*, 2nd edition, Microsoft Press, Redmond, WA, 2002.
4. Lipner, S., ``The Trustworthy Computing Security Development Lifecycle", 20th Annual Computer Security Applications Conference, Dec. 2004.
5. NCSP, "Improving Security across the Software Development Lifecycle," http://www.cyberpartnership.org/SDLCFULL.pdf, 2004.
6. Nancy, R., and McGraw G., "A Portal for Software Security," IEEE Security and Privacy, 2005.
7. McGraw, G. and Mead N.R. Engineering Security Into the Software Development Life Cycle. Crosstalk. The Journal of Defense Software Engineering, Oct 2005, Vol 18, No 10.
8. Viega, J., Security in the Software Development Lifecyle, IBM Rational Rose, October 2004.
9. Viega, J., Messier M, *Secure Programming Cookbook for C and C++*, O'Reilly, 2003.
10. Viega, J., McGraw G. *Building Secure Software*, Addison-Wesley, 2002.
11. Graff, M.G and Van Wyk, K.R. *Secure Coding, Principles, and Practices*, O'Reilly, 2002.
12. McGraw, G. *Software Security: Build Security In*, Addison-Wesley, 2006.
13. Howard, M. and Lipner, S. *The Security Development Lifecycle*, Microsoft Press, 2006.
14. Hoglund, G., McGraw G. *Exploiting Software: How to Break Code*, Addison-Wesley, 2004.
15. Alexander, I. Misuse Cases: Use Cases with Hostile Intent. IEEE Software, vol. 20, no. 1, pp. 58-66, 2003
16. Sindre, G., and Opdahl, A. Eliciting Security Requirements by Misuse CasesTOOLS-Pacific 2000. Proceedings. 37th International Conference on Technology of Object-Oriented Languages and Systems. pp. 120-131, 2000
17. Sindre, G., and Opdahl, A. Capturing Security Requirements through Misuse Cases. http://heim.ifi.uio.no/~nik/2001/21-sindre.pdf, 2001.
18. Myagmar, S., et al. Threat Modeling as a Basis for Security Requirements. Symposium on Requirements Engineering for Information Security (SREIS) in conjunction with 13th IEEE International Requirements Engineering Conference (RE). http://www.projects.ncassr.org/threatmodeling/sreis05.pdf, 2005
19. Kis, M. Information Security Antipatterns in Software Requirements Engineering. Conference

of Pattern Languages of Programs (PloP), 2002. http://jerry.cs.uiuc.edu/~plop/plop2002/final/mkis_plop_2002.pdf, 2002.

20. Liu, L. et al Security and Privacy Requirements Analysis within a Social Setting. Proceedings of the 11th IEEE International Requirements Engineering Conference 2003, pp. 151-161, 2003.
21. Swiderski, F. and Snyder, W., *Threat Modeling*, Microsoft Press, 2004.
22. Saltzer, J. and Schroeder, M., "The Protection of Information in Computing Systems", Proceedings of the IEEE 63(9), pp, 1278-1308, Sep. 1975.
23. Hafiz, M., ``The Security Architecture of qmail," Conference on Pattern Languages of Programs, Conference on Pattern Languages of Programs (PLoP 2004), ACM, 2004.
24. Provos, N. et al ``Preventing Privilege Escalation," 12th USENIX Security Symposium, Washington, DC, Aug. 2003.
25. Mullins, P. et al "Panel on Integrating Security Concepts into Existing Computer Courses", SIGCSE '02, Feb. 27 - Mar. 3, 2002, Covington, KY, 2002.
26. Davidson, M.A. "Leading by Example: the case for IT Security in Academia", Educause Review, Jan/Feb. 2005.
27. Whittaker, J. and Thompson, J., *How to Break Software Security*, Addison Wesley, 2003.
28. Security Wire Digest, Vol. 5, No. 59, August 7, 2003.
29. Carlo, G., et al , *Fundamentals of Software Engineering*, Prentice Hall, 1991.
30. Hopcroft, J. et al, *Introduction to Automata Theory, Languages, and Computation*, 2nd edition, Addison Wesley, 2001.
31. Wang, J.A., "Security Testing in Software Engineering Courses", *Proceedings of Frontiers in Education Conference*, Session F1C, IEEE Catalog Number 04CH37579C, ISBN: 0-7803-8553-5. October 2004.
32. Gilliam, D., et al, "Software Security Checklist for the Software Life Cycle," *Proceedings of the 12th IEEE International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE 2003)* pp. 243-248, June 2003.
33. OWASP, *A Guide for Building Secure Web Applications and Web Services*, *http://www.owasp.org/documentation/guide.html*, 2.0 Black Hat Edition, July 27, 2005.