Interdisciplinary Journal of Information, Knowledge, and Management Volume 5, 2010

Secure Software Engineering: A New Teaching Perspective Based on the SWEBOK

Manar Abu Talib
Zayed University, Abu Dhabi, UAE
manar.abutalib@zu.ac.ae

Adel Khelifi
Al Hosn University, Abu Dhabi, UAE
a.khelifi@alhosnu.ae

Leon Jololian
Zayed University, Abu Dhabi, UAE
leon.jololian@zu.ac.ae

Abstract

Lack of a suitable set of controls during the development life cycle of software will lead to mistakes in the requirements, design, or code of software and, therefore, result in significant security vulnerabilities. This paper proposes a software engineering course from the security perspective, which can be taught at both the undergraduate and graduate levels. It will prepare students to successfully cope with the technical challenges as well as the non technical issues associated with the software development process, while integrating security into each phase of the process. The course materials are derived from the Guide to the Software Engineering Body of Knowledge (SWEBOK) published by the IEEE Computer Society with the support of a consortium of industrial sponsors.

Keywords: Security, Software Engineering, SWEBOK, ISO 19759, Curriculum.

Introduction

Many authors (Graff & Van Wyk, 2002; Howard, 2005; Howard & LeBlanc, 2002; Lipner & Howard, 2005; Microsoft, 2009; Shumba, Walden, Ludi, Taylor, & Wang, 2006; Walden & Frank, 2006; Viega & McGraw, 2002, Viega & Messier, 2003) have discussed integration of the concept of security into the software life cycle; however, none of them has done so within the framework of the Software Engineering Body of Knowledge (SWEBOK). Moreover, from an academic point of view, few university software engineering courses or textbooks incorporate guidelines and practices related to "secure" software engineering. Most focus on securing only one phase of the development process, which is coding (Graff & Van Wyk, 2002; Howard & LeBlanc, 2002; Viega & Messier, 2003). From an industry point of view, current surveys indicate that we are far from being able to develop acceptably secure software systems, CERT (CERT, 2003; PricewaterhouseCoopers, 2004) having reported over 5,000 software vulnerabilities in 2005. One of the main reasons for this is that software engineers do not always have a strong background in computer security and lack expertise in secure software system development. In spite of this, in practice, they are asked to develop software systems that call for security features. Without appropriate methodologies and modeling languages to guide them during the development process, it is likely that they will fail to produce effective solutions (McDermott & Fox, 1999).

Articulating a body of knowledge is an essential step in the development of a

Editor: Eli Cohen

profession because it represents a broad consensus regarding the contents of the discipline. The IEEE Computer Society, with the support of a consortium of industrial sponsors, has published the Guide to the Software Engineering Body of Knowledge (SWEBOK). It has also gained international recognition as ISO Technical Report 19759. Although the concept of security is not explicitly referred to in it, the Guide describes generally accepted knowledge about software engineering, and its ten knowledge areas summarize basic concepts and include a list of references to detailed information. This paper takes from the Guide a summary of the guidelines and practices that can measurably reduce software requirements, as well as design and implementation defects, and improve the education of current and future software developers.

Our paper introduces a new way of teaching secure software engineering based on the SWEBOK Guide. This work differs from others by the following outcomes:

- The topic of security will be highlighted through the SWEBOK Guide.
- The summary of guidelines and practices are derived from the SWEBOK Guide to secure requirements analysis, design, implementation, and testing phases
- The proposed topics to be covered in the course during the academic term are provided based on the SWEBOK Guide.

Differences between our work and that of others are more detailed in the *Literature Review* section. The paper summarizes the guidelines and practices derived from the SWEBOK Guide in the *Proposed Guidelines and Practices based on SWEBOK* section. In the *Proposed Course Topics* section, it describes the detailed topics to be covered during the academic term, and in the section *Suggested Recommendations to Enhance SWEBOK Guide* we recommend some possible additions for SWEBOK 2010. The conclusions and our plans for future work are introduced in the last section.

Literature Review

Many authors have discussed the integration of security into the coding phase of the development process (Graff & Van Wyk, 2002; Howard & LeBlanc, 2002; Microsoft, 2009; Viega & McGraw, 2002, 2003). Howard and Leblanc (2002) described the best practices for writing secure code and stopping malicious hackers in their tracks, based on the knowledge of top security experts at Microsoft. Few university software engineering courses or textbooks have incorporated any secure software development topics into their courses. Frank, Walden, and Shumba (2006) provide valuable contributions to the design of a course in secure software engineering that will teach students how to incorporate security throughout the software development life cycle. For instance, Frank et al. introduced ten modules to cover the core topics in software security. Each module covers one or more class goals and will include both explanatory materials and assignments to give students the opportunity to apply their learning in a small context. The ten modules are: What is software security?, Threats and vulnerabilities, Risk management, Security requirements, Secure design principles and patterns, Secure programming through data validation, Secure programming through using cryptography securely, Code reviews and static analysis, Security testing, and, finally, Creating a software security program. It also explains the possibility of incorporating a team-based Web development project that students will work on throughout the semester to gain experience in applying security principles to a large-scale project.

The contribution of Shumba et al. (2006) to teaching on the secure development life cycle is illustrated in terms of the challenges and practices that have been introduced into the software engineering curriculum at five different universities. Each phase of the software development life cycle has been modified to incorporate security at one university at least. Shumba et al. provided a survey of practices involved in the secure development life cycle and described how these practices can be introduced into the software engineering curriculum. Each contributor discusses his or her experiences and challenges while integrating a specific stage of the life cycle 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 courses on secure embedded systems.

Articulating a Body of Knowledge is an essential step in the development of a profession because it represents a broad consensus regarding what a software engineering professional should know. Without such a consensus, no licensing examination can be validated, no curriculum can prepare an individual for an examination, and no criteria can be formulated for accrediting a curriculum. The development of the consensus is also a prerequisite to the adoption of coherent skills development and continuing professional education programs in organizations (Abran, Moore, Bourque, Dupuis, & Tripp, 2004). Therefore, this paper is proposing a course that can be taught at both the undergraduate and graduate levels based on such a broad consensus regarding what a software engineering professional should know.

Redwine (2006) has identified the additional body of knowledge necessary to develop, sustain, and acquire secure software beyond that normally required to produce software and ensure its quality. He picked a set of generic categories mapped to the categories used in a number of standards, curricula, and body-of-knowledge efforts. These are quite close to the categories used in the SWEBOK Guide – see Table 1.

Table 1: Comparison with the SWEBOK Guide (Samuel T. Redwine, 2006)

SWEBOK Guide	Software Assurance Document
	Threats and Hazards
	Fundamental Concepts and Principles
	Ethics, Law, and Governance
Software Requirements	Secure Software Requirements
Software Design	Secure Software Design
Software Construction	Secure Software Construction
Software Testing	Secure Software Verification, Validation and Evaluation
Software Quality	portions of Secure Software Engineering Management
Software Engineering Tools and Methods	Secure Software Tools and Methods
Software Engineering Process	Secure Software Processes
Software Engineering Management	Secure Software Engineering Management
	Acquisition of Secure Software
Software Maintenance	Secure Software Sustainment
Software Configuration Management	portions of Secure Software Engineering Management

What is the SWEBOK GUIDE?

The Guide to the Software Engineering Body of Knowledge (SWEBOK) (Abran et al., 2004) was established to promote a consistent view of software engineering worldwide. The build-up of a consistent worldwide view of software engineering was supported by a development process which engaged approximately 500 reviewers from 42 countries in the Stoneman phase (1998–

2001), which led to the Trial version, and over 120 reviewers from 21 countries in the Ironman phase (2003), which led to the 2004 version. That also helped provide a foundation for curriculum development, as well as for individual certification and licensing material.

Another objective of the SWEBOK Guide was to set a boundary for software engineering with respect to other disciplines, such as computer science, project management, computer engineering, and mathematics. The material that is recognized as belonging to this discipline is organized into the ten Knowledge Areas (KAs) listed in Table 2. Each of these KAs is treated as a chapter in the Guide.

Table 2: The SWEBOK Knowledge Areas (KAs) (Abran et al., 2004)

SWEBOK Knowledge Areas
Software requirements
Software design
Software construction
Software testing
Software maintenance
Software configuration management
Software engineering management
Software engineering process
Software engineering tools and methods
Software quality

In establishing a boundary, it is also important to identify what disciplines share that boundary and, often, a common intersection with software engineering. To this end, the Guide also recognizes eight related disciplines, as listed in Table 3 (Abran et al., 2004).

Table 3: The SWEBOK Related Disciplines (Abran et al., 2004)

Computer engineering	Project management
Computer science	Quality management
Management	Software ergonomics
Mathematics	Systems engineering

The challenges inherent in teaching secure software engineering are time limitations, lack of textbooks, and the immaturity of secure development methodologies and tools. However, the SWEBOK guidelines and practices for secure software development can be used without extensive alteration of the software engineering curriculum by selecting case studies and projects that have security relevance.

To the best of our knowledge, none has introduced the clear link between security and SWEBOK Guide in order to produce the secure software engineering course guidelines and practices. It is also to be noted that the following assumptions are made in order to run the secure software engineering course:

- The course is 14 weeks. Secure Software Maintenance is not covered in the course.
- Prerequisite courses such as introduction to software engineering and programming I are recommended for this course.

The Proposed Guidelines and Practices based on SWEBOK

The current SWEBOK Guide can serve as a reference for a secure software engineering course. The course overview is illustrated in Figure 1. The breakdown of the course constitutes the same first five Knowledge Areas described in the SWEBOK Guide, describing the decomposition of each Knowledge Area into subareas and topics. The five Knowledge Areas are: Software Requirements, Software Design, Software Construction, Software Testing, and Software Maintenance. However, here the term security has been highlighted throughout the Guide, and terms like secure software requirements, secure software design, secure software construction, secure software testing, and, finally, secure software maintenance have been added.

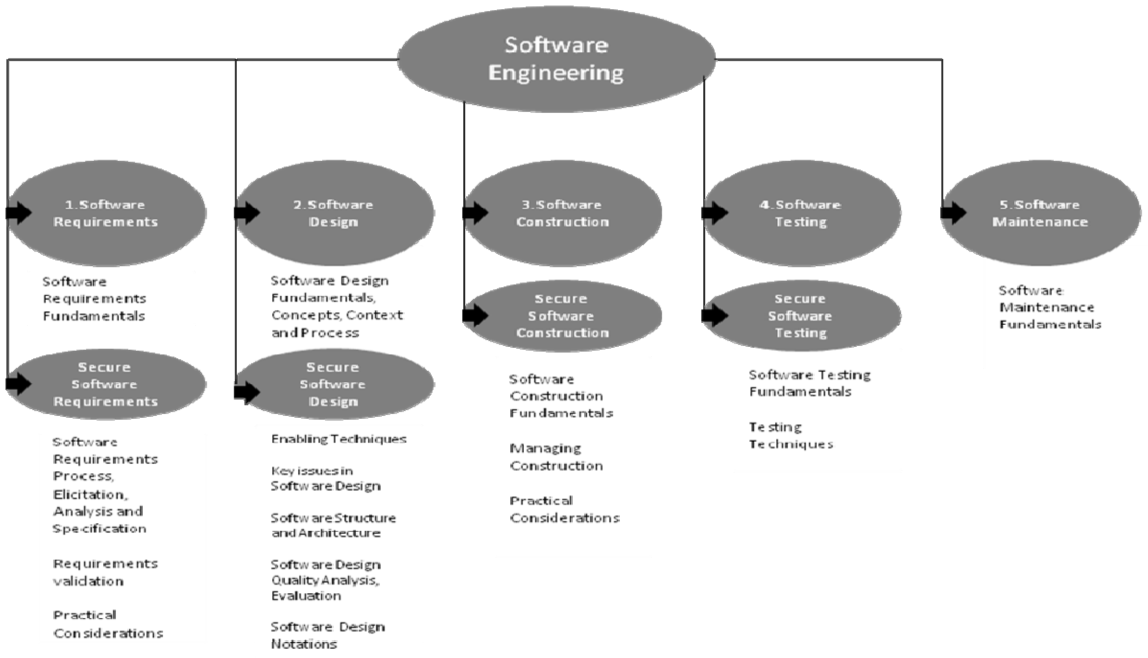
Software Engineering
1.Software Requirements
Software Requirements Fundamentals
Secure Software Requirements
Software Requirements Process, Elicitation, Analysis and Specification
Requirements validation
Practical Considerations
2.Software Design
Software Design Fundamentals, Concepts, Context and Process
Secure Software Design
Enabling Techniques
Key issues in Software Design
Software Structure and Architecture
Software Design Quality Analysis, Evaluation
Software Design Notations
3.Software Construction
Secure Software Construction
Software Construction Fundamentals
Managing Construction
Practical Considerations
4.Software Testing
Secure Software Testing
Software Testing Fundamentals
Testing Techniques
5.Software Maintenance
Software Maintenance Fundamentals

Figure 1: Course Overview

Software Requirements and Secure Software Requirements

In the software requirements phase, the students are able to define the requirements as properties that solve some real-world problem. Within the first two sub areas, "Software Requirements Fundamentals" and "Requirements Process," the students learn about the definitions of software requirements as well as the major types of requirements: product vs. process and functional vs. non-functional. Students will learn to describe process models, process actors, process support and management, and process quality and improvement. The second sub area, "requirement process," will introduce students to the first guideline on securing software requirements (as it is shown in Table 4), which is negotiating trade-offs that are both acceptable to the principal stakeholders and within budgetary, technical, regularity, and other constraints, that is because it will not be possible to perfectly satisfy the requirements of every stakeholder. The second guideline covers the link between the process activities identified in the process models and the issues of costs, human resources, training, and tools. The third guideline covers the improvement of the requirements process by using standards and models in terms of the cost and timeliness of a software product and of the customer's satisfaction with it.

The third sub area, “requirements elicitation,” is concerned with where software requirements come from and how the software engineer can collect them. It includes requirement sources and elicitation techniques. Learning these by themselves are practical guidelines that enhance the above proposed guidelines such as interviews, scenarios, prototypes, facilitated meetings and observations.

The fourth sub area, “requirements analysis,” is concerned with the process of analyzing requirements in order to detect and resolve conflicts between them and discover the bounds of the software and how it must interact with its environment. Requirements analysis includes classification, conceptual modeling, architectural design, and allocation of requirements, as well as requirements negotiation. Students learn many practical ways to secure software requirements that assist in understanding issues associated with modeling entities from the problem domain configured to reflect their real-world relationships and dependencies. The SWEBOK guide provides many examples of conceptual modeling, such as UML, formal modeling, IEEE Std 1320.1 for functional modeling, and IEEE Std 1320.2 for information modeling. Moreover, it refers to IEEE Std 1471-2000 for recommended practices for describing the architectural aspects of software-intensive systems. This standard suggests a multiple-viewpoint approach to describe the architecture of systems and their software items.

The fifth sub area, “requirements specification,” typically refers to the production of a document, or its electronic equivalent, that can be systematically reviewed, evaluated, and approved (Abran et al., 2004). The general rule is that notations should be used that allow the requirements to be described as precisely as possible. This rule serves as a guideline for students to consider in writing the software requirements specification document, keeping in mind a number of quality indicators. IEEE 1465 is a standard treating quality requirements in software packages. Students are also introduced to IEEE Std 830 for the production and content of the software requirements specification.

In addition, the last two sub areas, which are “requirements validation” and “practical considerations,” can be taught to students as security guidelines and practices. Many security vulnerabilities in software can be avoided if students are better equipped to recognize the security implications of their requirements choices. In requirements validation, they examine the requirements documents to ensure that they are defining the right system (that is, the system that the user expects). Requirements validation is subdivided into descriptions of the conduct of requirements reviews (IEEE Std 1028), prototyping, and model validation and acceptance tests (Abran et al., 2004).

The practical considerations sub area describes topics which need to be understood in practice. The first topic is the iterative nature of the requirements process. The next three topics are fundamentally about change management and the maintenance of requirements in a state which accurately mirrors the software to be built or that has already been built. This includes change management, requirements attributes, and requirements tracing. The final topic is requirements measurement (Abran et al., 2004).

Table 4 summarizes and highlights how security is applied in the software requirements phase within the SWEBOK Guide.

Table 4 Secure Software Requirements in SWEBOK Guide

SWEBOK Topics and References	Derived Guidelines
2.3. Process Support and Management	1. Negotiating the trade-offs. 2. Link between the process activities and the issues of costs, human resources, training and tools. 3. Improve the requirements process by using standards and models.
2.4. Process Quality and Improvement	
3. Requirements Elicitation 3.1. Requirements Sources	Practical guidelines such as interviews, scenarios, prototypes, facilitated meetings and observations.
3.2. Elicitation Techniques	
4. Requirements Analysis 4.1. Requirements Classification	Practical guidelines and standards such as UML, formal modeling, IEEE Std 1320.1 for functional modeling, IEEE Std 1320.2 for information modeling and IEEE Std 1471-2000.
4.2. Conceptual Modeling	
4.3. Architectural Design and Requirements Allocation	
4.4. Requirements Negotiation	
5. Requirements Specifications 5.1. The System Definition Document	Standards such as IEEE Std 830 and IEEE 1465.
5.2. System Requirements Specification	
5.3 Software Requirements Specification	
6. Requirements Validation 6.1. Requirements Reviews	1. IEEE 1028 provides guidance on conducting requirements review. 2. Prototyping for validating the software engineer's interpretation of the software requirements. 3. Static analysis to verify that communication paths exist between objects. 4. Perform acceptance testing.
6.2. Prototyping	
6.3. Model Validation	
6.4. Acceptance Tests	
7. Practical Considerations 7.1. Iterative Nature of the Requirements Process	1. Go through a defined review and approval process, and apply careful requirements tracing, impact analysis and software configuration management. 2. IEEE Std 14143.1 defines the concept of Functional Size Measurement to evaluate the size of change in requirements or in estimating the cost of a development or maintenance task.
7.2. Change Management	
7.3. Requirements Attributes	
7.4. Requirements Tracing	
7.5. Measuring Requirements	

Software Design and Secure Software Design

In software design phase, the students learn how to define the architecture, components, interfaces, and other characteristics of a system or component. They accomplish this through learning first about the software design fundamentals, which form an underlying basis to the understanding of the role and scope of software design. These fundamentals are: general software concepts, the context of software design, the software design process, and the enabling techniques for software design (Abran et al., 2004).

Software design assumptions and choices that determine how the software will operate and how different modules/components will interact at this phase should be analyzed and adjusted to minimize the exposure of those functions and interfaces to attackers. Therefore, to secure such software design phase and reduce design defects, they have to learn some enabling techniques that are key notions considered fundamental to many different software design approaches and concepts such as IEEE1016-98 for recommended practice for software design descriptions. Moreover, the students deal with number of key issues when designing software such as how to structure and organize the interactions with users and the presentation of information i.e. using the Model-View-Controller approach. They also deal with other issues, for example concurrency, control and handling of events, distribution of components, error and exception handling, and fault tolerance and data persistence. All of these are derived guidelines and practices for securing software design.

Students are introduced to the architectural structures and viewpoints, architectural styles, design patterns, and families of programs and frameworks as best practices of securing software design. They learn different ideas about software design at different levels of abstraction: different architectural styles (such as general structure, distributed systems, interactive systems, and adaptable systems) and different design patterns (such as creational patterns, structural patterns, and behavioral patterns).

In addition, the students know some basics about the design notations as tools to be used in producing accurate pictures about the software besides capturing some quality attributes. The students will be introduced to ISO9126-01 for software engineering – product quality and ISO15026-98 for system and software integrity levels. They also will be introduced to various tools and techniques such as software design reviews, static analysis, and simulation and prototyping. They will briefly learn about function-oriented design measures and object-oriented design measures (Abran et al., 2004). These are another set of guidelines and practices for securing software design.

The final derived guideline is understanding some general strategies to help guide the design process as a means of transferring knowledge and as a common framework for teams of software engineers such as divide-and-conquer and stepwise refinement, top-down vs. bottom-up strategies, data abstraction and information hiding, use of heuristics, use of patterns and pattern languages, use of an iterative and incremental approach. Table 5 highlights how security is applied in software design phase within SWEBOK Guide.

Table 5 Secure Software Design in SWEBOK Guide

SWEBOK and Topics and References	Derived Guidelines
1.4. Enabling Techniques	Learning enabling techniques that are key notions considered fundamental to many different software design approaches and concepts such as IEEE1016-98 for recommended practice for software design descriptions.
1.4.1. Abstraction	
1. 4.2. Coupling and Cohesion	
1. 4.3. Decomposition and modularization	
1.4. 4. Encapsulation/information hiding	
1. 4.5. Separation of interface and implementation	
1. 4.6. Sufficiency, completeness and primitiveness	

2. Key Issues in Software Design	Dealing with number of key issues when designing software such as: 1. How to decompose, organize and package software components. 2. How to structure and organize the interactions with users and the presentation of information i.e. using the Model-View-Controller approach.
2.1. Concurrency	
2.2. Control and Handling of Events	
2.3. Distribution of Components	
2.4. Error and Exception Handling and Fault Tolerance	
2.5. Interaction and Presentation	
3. Software Structure and Architecture 3.1. Architectural Structures and Viewpoints	Learning about different ideas about software design at different levels of abstraction: 1. Different architectural styles such as general structure, distributed systems, interactive systems, adaptable systems, etc. 2. Different design patterns such as creational patterns, structural patterns and behavioral patterns. 3. Different families of programs and frameworks.
3.2. Design Patterns	
3.3. Families of Programs and Frameworks	
4. Software Design Quality Analysis and Evaluation	Learning about quality that is related to software design: 1. ISO9126-01 for software engineering – product quality and ISO15026-98 system and software integrity levels. 2. Through various tools and techniques such as software design reviews, static analysis and simulation and prototyping. 3. Function-oriented design measures and object-oriented design measures.
4.1. Quality Attributes	
4.2.Quality Analysis and Evaluation Techniques	
4.3. Measures	
5. Software Design Notations 5.1. Structural Descriptions (static view)	Learning many notations and languages to represent software design artifacts.
5.2. Behavioral Descriptions (dynamic view)	
6. Software Design Strategies and Methods	Learning various general strategies to help guide the design process as a means of transferring knowledge and as a common framework for teams of software engineers such as divide-and-conquer and stepwise refinement, top-down vs. bottom-up strategies, data abstraction and information hiding, use of heuristics, use of patterns and pattern languages, use of an iterative and incremental approach.
6.1. General Strategies	
6.2. Function-Oriented (Structured) Design	
6.3. Object-Oriented Design	
6.4. Data-Structure-Centered Design	
6.5. Component-Based Design (CBD)	
6.6. Other Methods	

Software Construction and Secure Software Construction

Software construction refers to the detailed creation of working, meaningful software through a combination of coding, verification, unit testing, integration testing, and debugging. Students will learn about the fundamentals of software such as minimizing complexity, anticipating change, and constructing for verification. This practical assistance for secure software construction teaches students how to create code that is simple and readable by using various techniques as well as construction standards, such as those of the IEEE and ISO. They secure the construction phase by anticipating changes, using techniques such as communication methods, programming languages, platforms, and tools.

Moreover, the students will be provided with an overview of construction design, construction languages, coding, construction testing, reuse, construction quality, and integration. The derived guidelines, practices and standards are summarized in Table 6.

Table 6 Secure Software Construction in SWEBOK Guide

SWEBOK Topics and References	Derived Guidelines
1. Software Construction Fundamentals 1.1. Minimizing Complexity 1.2. Anticipating Change 1.3. Constructing for Verification 1.4. Standards in Construction	1. Learning how to create code that is simple and readable by using various techniques as well as standards in construction such as the IEEE and ISO. 2. Anticipating changes by using techniques such as communication methods, programming languages, platforms and tools. 3. Constructing for verification by following code reviews, unit testing, organizing code to support automated testing and restricted use of complex or hard to understand language structures, among others.
2. Managing Construction 2.1. Construction Models 2.2. Construction Planning 2.3. Construction Measurement	1. Learning numerous models to develop software such as the waterfall model, staged-delivery model, evolutionary prototyping, extreme programming and scrum. 2. Defining the order in which components are created and integrated, the software quality management processes, the allocation of task assignments to specific software engineers, and the other tasks, according to the chosen model. 3. Measuring for purpose of managing construction, ensuring quality during construction, improving the construction process, etc.
3. Practical considerations 3.1. Construction Design 3.2. Construction Languages 3.3. Coding 3.4. Construction Testing 3.5. Reuse 3.6. Construction Quality 3.7 Integration	1. Learning various techniques and practical considerations to enhance the construction activities and reduce the gap between the time at which faults are inserted into code and the time those faults are detected. 2. Using many standards such as IEEE Std 829-1998, IEEE Standard for Software Test Documentation, IEEE Std 1008-1987, IEEE Standard for Software Unit Testing, IEEE Std 1517-1999, IEEE Standard for Information Technology-Software Life Cycle Processes- Reuse Processes and ISO/IEC 12207:95, Standard for Information Technology-Software Life Cycle Processes.

Software Testing and Secure Software Testing

Software testing consists of the dynamic verification of the behavior of a program on a finite set of test cases, suitably selected from the usually infinite execution domain, against expected behavior. At the beginning of this phase, students learn the software testing fundamentals. First, the testing-related terminology is presented, then key issues of testing are described, and, finally, the relationship of testing to other activities is covered (Abran et al., 2004).

The main objectives of securing software testing, according to Jarzombek & Goertzel (2006), are: 1) Detection of security defects, coding errors, and other vulnerabilities, including those generated from complex relationships among functions and those that exist in obscure areas of code, such as dormant functions; 2) demonstration of continued secure behavior when subjected to attack patterns; and 3) verification that the software consistently exhibits its required security properties and functional constraints under both normal and hostile conditions. To teach them to meet these objectives, the students will learn some test techniques. The first category includes tests based on the tester's intuition and experience. A second group comprises specification-based techniques, followed by code-based techniques, fault-based techniques, usage-based techniques, and techniques related to the nature of the application (Abran et al., 2004).

A discussion of how to select and combine the appropriate techniques is also presented as a guideline to secure the testing phase. The course will cover some test-related measures, grouped into those related to the evaluation of the program under test and the evaluation of the tests performed. Brief practical considerations will be introduced to the students.

The Table 7 highlights how security is applied in the software testing phase within the SWEBOK Guide.

Table 7 Secure Software Testing in SWEBOK Guide

SWEBOK Topics and References	Derived Guidelines
1.2. Key issues 1.2.2. Testing effectiveness/Objectives for testing 1.2.3. Testing for defect identification	Detection of security defects, coding errors, and other vulnerabilities.
3. Test Techniques 3.1. Based on the software engineer's intuition and experience	Learning various techniques and practical considerations such as 1. ad hoc testing 2. exploratory testing 3. equivalence partitioning 4. boundary-value analysis 5. decision table 6. finite-state machine-based 7. testing from formal specifications 8. random testing 9. control-flow-based criteria 10. data flow-based criteria 11. reference models for code-based testing (flowgraph, call graph)
3.2. Specification-based techniques	
3.3. Code-based techniques	

Software Maintenance

Once in operation, anomalies are uncovered, operating environments change, and new user requirements surface. The maintenance phase of the life cycle commences upon delivery, but maintenance activities are performed much earlier. Students are introduced to the software maintenance fundamentals. They learn some definitions and terminology, the nature of maintenance, the need for maintenance, the majority of maintenance costs, the evolution of software, and the categories of maintenance.

Proposed Course Topics

As was discussed in the previous section, the proposed topics are summarized in Table 8. This can be run as a 14-week course and can be offered as an undergraduate or graduate course for students who have little background in software engineering or programming.

Table 8 Proposed Topics for a Secure Software Engineering Course

Week #	Topic	Detailed subtopics	SWEBOK Reference
#1	Software Engineering	**1. What is Software Engineering?** **2. What is Secure Software Engineering?** 2.1 Threats and Vulnerabilities	Chapter 1
#2	Software Requirements	**1. Software Requirements Fundamentals** **2. Requirements Process** **3. Requirements Elicitation**	Chapter 2
#3-#4	Secure Software Requirements	**4. Requirements Analysis** **5. Requirements Specification** **6. Requirements validation** 6.1 Requirements Reviews 6.2 Prototyping 6.3 Model Validation 6.4 Acceptance Tests **7. Practical Considerations** 7.1 Iterative Nature of the Requirements Process 7.2 Change Management 7.3 Requirements Attributes 7.4 Requirements Tracing 7.5 Measuring Requirements	Chapter 2 Sections 6&7
#5	Software Design	**1. Software Design Fundamentals**	Chapter 3
#6-#8	Secure Software Design	**1. Enabling Techniques** 1.1. Abstraction 1.2. Coupling and cohesion 1.3. Decomposition and modularization 1.4. Encapsulation/information hiding 1.5. Separation of interface and implementation 1.6. Sufficiency, completeness and primitiveness **2. Key Issues in Software Design** 2.1. Concurrency 2.2. Control and Handling of Events 2.3. Distribution of Components 2.4. Error and Exception Handling and Fault Tolerance 2.5. Interaction and Presentation 2.6. Data Persistence **3. Software Structure and Architecture** 3.1. Architectural Structures and Viewpoints 3.2. Design Patterns (overview) 3.3. Families of Programs and Frameworks **4. Software Design Quality Analysis and Evaluation** 4.1. Quality Attributes 4.2. Quality Analysis and Evaluation Techniques 4.3. Measures **5. Software Design Notations** **6. Software Design Strategies and Methods**	Chapter 3

#9	Software Construction (Implementation Phase)	**1. Software Construction Fundamentals**	Chapter 4
#10-#11	Secure Software Construction	**1. Software Construction Consideration** 1.1. Minimizing Complexity 1.2. Anticipating Change 1.3. Constructing for Verification 1.4. Standards in Construction **2. Managing Construction** 2.1. Construction Models 2.2. Construction Planning 2.3. Construction Measurement	Chapter 4 Sections 1&2
#12	Software Testing	**1. Software Testing Fundamentals** **2. Testing Techniques** **3. Secure Software Testing (Brief)**	Chapter 5
#13	Software Maintenance	**1. Software Maintenance Fundamentals** 1.1. Definitions and Terminology 1.2. Nature of Maintenance 1.3. Need for Maintenance 1.4. Majority of Maintenance 1.5. Evolution of Software 1.6. Categories of Maintenance **2. Key Issues in Software Maintenance**	Chapter 6

Suggestions for Enhancing the SWEBOK Guide

Previous sections have shown that the current SWEBOK Guide can serve as a reference for a secure software engineering course. The SWEBOK Guide can be upgraded by highlighting the term "security" throughout the Guide, as well as adding terms such as secure software requirements, secure software design, secure software construction, secure software testing, and finally secure software maintenance (refer to the section on the Proposed Guidelines and Practices based on SWEBOK). In this section, we recommend some additions that could be taken into consideration for SWEBOK 2010. They are summarized as follows:

- Introduce definitions for software security, security threats, and security vulnerabilities, as well as explain how software security is different from security features and how security vulnerabilities arise from poor software engineering practices (Walden & Frank, 2006).
- Define significant insider threat vulnerabilities that can be introduced during all phases of the software development life cycle (CERT, 2008).
- Include security engineering activities, such as security requirements elicitation and definition, secure design based on design principles for security, use of static analysis tools, secure reviews and inspections, and secure testing methods. A good source of information about secure engineering activities is the Department of Homeland Security (DHS)'s Build Security In web site (https://buildsecurityin.us-cert.gov/portal/). The Systems Security Engineering Capability Maturity Model (SSE-CMM) is a process model that can be also used to improve and assess the security engineering capability of an organization. The SSE-CMM provides a comprehensive framework for evaluating security engineering practices against the generally accepted security engineering principles. By defining such a framework, the SSE-CMM provides a way to measure and improve performance in the

application of security engineering principles (Redwine & Davis, 2004). The SSE-CMM has been adopted as the ISO/IEC 21827 standard. Further information about the model is available at http://www.sse-cmm.org

- Include security assurance activities, such as verification, validation, expert review, artifact review, and evaluations. Security assurance usually also includes activities for the requirements, design, implementation, testing, release, and maintenance phases of the software development life cycle (NASA, 1989).
- Include organizational activities, such as organizational policies, senior management sponsorship and oversight, establishing organizational roles, and other organizational activities that support security. Project management activities include project planning and tracking, and resource allocation and usage to ensure that the security engineering, security assurance, and risk identification activities are planned, managed, and tracked.
- Introduce risk management as a means to evaluate the risks to an application by identifying its assets, along with threats to the confidentiality, integrity, or availability of those assets. Risk management is also a means to rank those risks to determine which risks need to be mitigated and which risks can be accepted (Walden & Frank, 2006). Security risk is also addressed at the Department of Homeland Security (DHS) Build Security in web site (https://buildsecurityin.us-cert.gov/portal/).
- Build security into the early stages of the software development life cycle by addressing security requirements and prioritizing them. Then, the requirements elicitation and analysis can take place on this set of security requirements. The National Institute of Standards and Technology (NIST) reports that software that is faulty in terms of security and reliability costs the economy $59.5 billion annually in breakdowns and repairs. The costs of poor security requirements make it apparent that even a small improvement in this area will generate a high value. The SWEBOK Guide can refer to the Security Quality Requirements Engineering (SQUARE) methodology, which consists of nine steps that generates a final deliverable of categorized and prioritized security requirements (Mead, Hough, & Stehney, 2005). Papers on incorporating security requirements engineering into the dynamic systems development method and into the rational unified process have also been presented at conferences (Mead, Venkatesh, & Padmanabhan, 2008; Mead, Venkatesh, & Zhan, 2008).
- Include more coding standards and best practices to secure software construction and stop malicious hackers in their tracks. For example, ISO/IEC TR 24772, "Guidance to Avoiding Vulnerabilities in Programming Languages through Language Selection and Use" (ISO/IEC TR 24772, 2008), the CERT C Secure Coding Standard, with that has more than 220 contributors and reviewers participating in its standard's development (Seacord, 2005), as well as, Secure Coding in C and C++ (CERT, 2008, Seacord, 2008).
- Introduce variety into code review types, ranging from informal peer reviews to formal inspection and static analysis tools. The Secure Coding Initiative (SCI) is working with industry partners, such as LDRA and Fortify Software, and with research partners, such as JPCERT and Lawrence Livermore National Laboratory, to enhance existing source code analysis tools (CERT 2008).
- Include more information about Software Engineering process management (Davis, 2005), such as Capability Maturity Model Integration (CMMI), the Federal Aviation Administration integrated Capability Maturity Model (FAA-iCMM) (FAA 01), Trusted CMM/Trusted Software Methodology (T-CMM/TSM) (Kitson, 1995), Systems Security Engineering Capability Maturity Model (SSE-CMM) (Redwine & Davis, 2004), or

ISO/IEC 21827, Microsoft's Trustworthy Computing Security Development Lifecycle (SDL) (Lipner & Howard, 2005), Team Software Process for Secure Software Development (TSP), Correctness by Construction (Hall & Chapman, 2002), Agile Methods, Common Criteria for Information Technology Security Evaluation or ISO/IEC 15408.

- Include other key standards and methods that apply to developing secure software, such as ISO/IEC 15288 for System Life Cycle Processes and Cleanroom Software Engineering (Linger, 1994; Mills & Linger, 1987).

Conclusions and Future Work

This paper describes the integration of security into the Software Engineering curriculum as per the SWEBOK Guidelines. It summarizes the secure development guidelines and practices into each phase of the life cycle based on the SWEBOK Guide. It proposes the topics to be covered during an academic term. Further work is required to apply the proposed secure requirements analysis, design, implementation, and testing guidelines and practices into a single semester software engineering course at the graduate level at Zayed University in the UAE. The instructor will evaluate the course through quantitative measures in order to estimate the knowledge gained on securing software engineering. Also planned is a research to update the secure software engineering course topics and the guidelines upon release of the 2010 version of the SWEBOK Guide.

Acknowledgment

Thanks to Dr. Alain Abran and Dr. Pierre Bourque, who have reviewed the paper and provided significant feedback.

References

Abran, A., Moore, J.W., Bourque, P., Dupuis, R., and Tripp, L.L.: Software Engineering Body of Knowledge. Los Alamitos: IEEE Computer Society Press, 2004 edition.

CERT. (2003). *Secure systems.* Retrieved from CERT: www.cert.org

CERT. (2008). *CERT's podcasts: Security for business leaders: Show notes.* Retrieved January 31, 2010, from CERT Podcast Series http://www.cert.org/podcast/notes/20080304cappelli-notes.html

Davis, N. (2005). *Secure software development life cycle processes: A technology scouting report.* Pittsburgh, Pennsylvania, USA: Software Engineering Institute, Carnegie Mellon University.

Frank, C., Walden, J., & Shumba, R. (2006). SIGCSE 2006 birds of a feather: Secure software engineering. *37th SIGCSE Technical Symposium on Computer Science Education* (p. 573). Houston, Texas, USA.

Graff, M., & Van Wyk, K. R. (2002). *Secure coding, principles, and practices.* O'Reilly.

Hall, A., & Chapman, R. (2002). Correctness by construction: Developing a commercial secure system. *IEEE Software*, 18-25.

Howard, M. (2005, November). *A look inside the security development lifecycle at Microsoft.* Retrieved September 30, 2009, from MSDN: http://msdn.microsoft.com/en-us/magazine/cc163705.aspx

Howard, M., & LeBlanc, D. (2002). *Writing secure code* (2nd ed.). Redmond, WA: Microsoft Press.

ISO/IEC TR 24772. (2008). *Information technology — Programming languages — Guidance to avoiding vulnerabilities in programming languages through language selection and use.* Geneva: ISO.

Jarzombek, J. & Goertzel, K. M. (2006). Security in the software life cycle. *CrossTalk: The Journal of Defense Software Engineering*. Retrieved from http://www.stsc.hill.af.mil/Crosstalk/2006/09/0609JarzombekGoertzel.html

Kitson, D. H. (1995). A tailoring of the CMM for the trusted software domain. *Proceedings of the Seventh Annual Software Technology Conference.* Salt Lake City, Utah.

Linger, R. C. (1994). Cleanroom process model. *IEEE Software, 11*(2), 50-58.

Lipner, S., & Howard, M. (2005, March). *The trustworthy computing security development lifecycle.* Retrieved September 30, 2009, from MSDN: http://msdn.microsoft.com/en-us/library/ms995349.aspx

McDermott, J., & Fox, C. (1999). Using abuse case models for security requirements analysis. *15th Annual Computer Security Applications Conference (ACSAC '99)* (p. 55). Scottsdale, AZ, USA: IEEE Computer Society.

Mead, N. R., Hough, E., & Stehney, T. (2005). *Security Quality Requirements Engineering (SQUARE) Methodology (CMU/SEI-2005-TR-009).* Pittsburgh: Software engineering Institute, Carnegie Mellon University.

Mead, N. R., Venkatesh, V., & Padmanabhan, D. (2008). Incorporating security requirements engineering into the dynamic systems development method. *COMPSAC (International Computer Software and Applications Conference), IWSSE Workshop (International Workshop on Security and Software Engineering), July 28, 2008, Turku, Finland. IEEE Computer Society*, 949–954.

Mead, N. R., Venkatesh, V., & Zhan, J. (2008). Incorporating security requirements engineering into the rational unified process. *International Conference on Information Security and Assurance (ISA), Busan, Korea, April 26–28, IEEE Computer Society*, 537–542.

Microsoft. (2009, May 19). *The security development lifecycle: Making secure code easier.* Retrieved September 30, 2009, from MSDN Blog: http://blogs.msdn.com/sdl/archive/2009/05/19/making-secure-code-easier.aspx

Mills, H., & Linger, R. C. (1987). Cleanroom software engineering. *IEEE Software, 4*(5), 19-25.

NASA. (1989, September). *Software assurance guidebook, NASA-GB-A201*. Retrieved January 31, 2010, from http://satc.gsfc.nasa.gov/assure/agb.txt

PricewaterhouseCoopers. (2004). *Information security breaches survey.* Retrieved January 31, 2010, from PricewaterhouseCoopers: http://www.pwc.co.uk/pdf/dti_technical_report_2004.pdf

Redwine, S. T. (2006). Software assurance: A guide to the common body of knowledge to produce, acquire, and sustain secure software. *US Department of Homeland Security*.

Redwine, S. T., & Davis, N. (2004). Processes to produce secure software: Towards more secure software. Retrieved from http://www.criminal-justice-careers.com/resources/Software+Pro.pdf

Seacord, R. C. (2005). *Secure coding in C and C++*. Boston: Addison- Wesley Professional.

Seacord, R. C. (2008). *The CERT C secure coding standard.* Boston: Addison-Wesley Professional.

Shumba, R., Walden, J., Ludi, S., Taylor, C., & Wang, J. A. (2006). Teaching the secure development lifecycle: Challenges and experiences. *10th Colloquium for Information Systems Security Education.*

Viega, J. & McGraw, G. (2002). *Building secure software.* Addison-Wesley.

Viega, J.& Messier M. (2003). *Secure programming cookbook for C and C++*. O'Reilly.

Walden, J., & Frank, C. E. (2006). Secure software engineering teaching modules. *3rd Annual Conference on Information Security Curriculum Development.*

Biographies

Manar Abu Talib is an assistant professor at the Information Technology College of Zayed University in the UAE. She holds a PhD in Computer Science and Software Engineering (2007) from Concordia University in Montreal, Canada. Manar is a researcher in the area of software engineering with substantial experience and knowledge in conducting research in software measurement analysis, design, and testing. She has an impressive list of papers to her credit, which have most recently been accepted by journals in Canada, Italy, and Germany.

Adel Khelifi is an assistant professor in the Software Engineering department and director of IT at ALHOSN University, UAE. He has had an impressive career, most recently working as a lecturer at the École de technologie supérieure in Canada and, previously, for the United Nations MSF in Canada, for Canada's Ministry of Citizenship and Immigration and for the Ministry of Finance in Tunisia. Currently, he is involved in developing software engineering course content, including software quality, software testing, and software maintenance. As a Canadian ISO member in software engineering, Dr. Khelifi is contributing to the development of software measurement standards.

Leon Jololian joined Zayed Unversity in 2006. He has been the acting Dean, then Dean of the College of IT since 2007. Prior to joining Zayed University, Dr. Jololian served on the faculty of several universities in the US, including New Jersey City University, University of Alabama at Birmingham, and New Jersey Institute of Technology. Dr. Jololian has considerable research publications and is a holder of a US patent. He has helped establish and launch the graduate program in the College of IT at Zayed University which offers an MS degree in information security with specialization in cyber security. Under his leadership, the College of IT is going through the process of seeking ABET accreditation for its undergraduate IT program.