TOWSON UNIVERSITY

COLLEGE OF GRADUATE STUDIES AND RESEARCH

AN INTEGRATED SECURITY CURRICULUM MODEL FOR UNDERGRADUATE COMPUTER SCIENCE AND INFORMATION SYSTEMS MAJORS

by

Blair Taylor

A thesis
presented to the faculty of
Towson University
in partial fulfillment
of the requirements for the degree
Doctor of Science

May 2008

Towson University
Towson, Maryland 21252

TOWSON UNIVERSITY
COLLEGE OF GRADUATE STUDIES AND RESEARCH

THESIS APPROVAL PAGE

This is to certify that the thesis prepared by Blair Taylor, entitled AN INTEGRATED SECURITY CURRICULUM MODEL FOR UNDERGRADUATE COMPUTER SCIENCE AND INFORMATION SYSTEMS MAJORS, has been approved by this committee as satisfactory completion of the requirement for the degree of Doctor of Science in Applied Information Technology in the department of Computer and Information Sciences.

_____ _____
Chair, Thesis Committee Date

_____
Print Name

_____ _____
Committee Member Date

_____
Print Name

_____ _____
Committee Member Date

_____
Print Name

_____ _____
Committee Member Date

_____
Print Name

_____ _____

Dean, College of Graduate Studies and Research Date

_____

Print Name

ABSTRACT

AN INTEGRATED SECURITY CURRICULUM MODEL FOR UNDERGRADUATE COMPUTER SCIENCE AND INFORMATION SYSTEMS MAJORS

Blair Taylor

Security education, especially for undergraduates, has been identified as important to changing problematic security practices. However, current security efforts in education are primarily in the form of tracks or specialized courses, which only reach a subset of students and occur after students have established a foundation of coding techniques. This project aimed to develop, implement, and evaluate an integrated security curriculum model for students in the Computer and Information Sciences department at Towson University. Security modules were developed and subsequently delivered during spring 2007 and fall 2007 across selected sections of the core courses: CS0, CS1, and CS2. The modules were laboratory-based to allow seamless adoption and integration and included the innovative use of checklists to promote active learning and encourage critical thinking. A two group control group experimental design was employed using pretests and posttests for evaluation. A significant improvement in security knowledge was demonstrated ($p = .005$) indicating that security lab modules are an effective way to teach secure coding and design principles to more students, earlier in their studies, with minimal impact on existing curricula.

Towson University, as a designated National Center of Academic Excellence in Information Security and Assurance Education, is the ideal platform to pioneer a "security across the curriculum" approach. The integrated security curriculum presented in this research complements our undergraduate security track for computer science majors and serves as a model for future integration.

Table of Contents

List of Tables

List of Figures

1. Introduction

Computer security is a national crisis; the number of attacks and software vulnerabilities continue to rise (CERT, 2007). In response, many businesses, such as Microsoft, have adopted a Secure Development Lifecycle (SDL) that incorporates security at all levels of software development. Colleges and universities have reacted by developing security tracks and specialized security courses. While security tracks are effective at training security experts, they only reach a subset of the students and occur after students have established a foundation of coding techniques. Consequently, the majority of undergraduate computing students learn programming and design with little regard to security issues. In other words, colleges and universities are doing too little, too late.

To ensure all undergraduate computing graduates have the foundation and skills necessary to develop secure information systems, education must infuse secure coding and design principles early and often in the learning process. Increasingly, security education experts are recognizing that security can no longer be an afterthought, but instead must be seamlessly integrated or threaded across the entire computer science curriculum, beginning with the foundation courses and re-enforced throughout all students' course of study (Conklin & Dietrich, 2006; Davis & Dark, 2003; Graf & van Wyck, 2003; Hoglund & McGraw, 2004; Howard & LeBlanc, 2003; Irvine et al., 1998; Perrome et. al., 2005; Viega & McGraw, 2002).

Security integration is a daunting task: implementation details and models are lacking; the existing undergraduate computing curriculum is already over-extended and

challenging; there is a shortage of security-trained faculty; and security issues are omni-present and dynamic.

This thesis introduces a security integration module that uses security labs and checklists to integrate security throughout the computer science and computer information systems curriculum. The literature review to follow will demonstrate the need for comprehensive security integration beyond security tracks. The purpose of this research is to show the effectiveness of the security injection modules in reaching more students earlier in their studies. We will also examine the extensibility of the modules for future classes and other institutions.

1.1 Background

Towson University is a designated National Center of Academic Excellence (CAE) in Information Security and Assurance Education, having met all five federal requirements in information assurance education. In 2002, we introduced a specialized security track for Computer Science (CS) majors. Results from the track have been positive (Azadegan et. Al, 2006); it has attracted highly qualified students and initial industry feedback has indicated that the graduates are well prepared. However, minority and female representation is poor. Additionally, we feel that critical security concepts covered exclusively in the track classes are essential to other specialty areas. For example, a project manager studying CS, but not opting for the track, needs to learn about risk management.

In order for integration to be feasible, the impact on the already overextended CS and CIS curriculum must be as seamless as possible. With this in mind, we have

identified the laboratory as a place to introduce security modules which include security-related labs, checklists, and analytical and research questions.

1.2 Statement of the Problem

Currently, security is not well integrated into the undergraduate computer or information science curriculum. Many educators lack knowledge in security (PITAC, 2005) and most security efforts at the undergraduate level are in the form of specialized security classes or tracks. Security classes address particular topics; security tracks are effective at training security experts. Both approaches require sufficient resources, including trained faculty. Additionally, both only address a subset of the students and are generally offered after students have established a foundation of coding techniques. The result is that the majority of computing students are taught programming and design with little regard to security issues.

1.3 Purpose of Research

To complement our security track, we have developed a plan to infuse security principles throughout our entire undergraduate CS and Computer Information Systems (CIS) programs. Our objectives are to broaden the scope of our security initiative to include all computing majors and to introduce security concepts from the first course. The ultimate goal is comprehensive integration, including all courses in the CS and CIS curriculum. Our model is designed to be replicable for inclusion at other institutions and efforts have begun to foster collaboration and grant opportunities.

1.4 Significance

Software security is a critical topic that must be addressed by the educational

community. This research complements our security track by integrating security principles throughout the undergraduate curriculum. Pioneering this effort to better prepare future software professionals in secure coding fundamentals addresses the nationwide security crisis, strengthens Towson's role as a CAE, and places our department at the forefront of security education.

1.5 Research Design

This research study began by integrating security modules across sections of CS0, CS1, and CS2. Security lab modules were developed as part of the research and an assessment test was created for pre and post-evaluation. Testing was administered on-line across all sections of the core courses. In spring 2007, we piloted security integration across all three sections of CS0 and two of five sections in CS1. In fall 2007, we repeated the study and added one section of CS2. We conducted security pretests and posttests across all sections and compared the results of the integration effort.

In addition to the core courses, materials have been developed for upper level courses from each major, these courses include: Programming Languages: Design and Implementation (PROG), System Analysis and Design (SYSANAL), and Database Management Systems (COSCDB and CISDB). This semester, spring 2008, integration is beginning in these classes.

The cornerstones to integration include:

1. The Secure Development Lifecycle
2. Software Security Touchpoints

- Program defensively
- Defense in depth
- Hiding information
- Least privilege
- All input is evil
- Assume the impossible
- Deny by default
- Design by contract
- Basic understanding of hostile environment
- Selection of Strong Password
- Strong typing
- Avoid variable length fields
- Testing
- Graceful degradation
- All errors have potential for vulnerability

3. Risk Analysis- Also referred to as threat modeling, this is another important security concept identified as integral to the SDL (McGraw, 2006). Understanding threats is critical to building a secure system (Swiderski & Snyder, 2004). The risk analysis process is well defined, and begins with identifying and ranking threats and vulnerabilities (Perrone, Aburdene, & Meng, 2005).

1.6 Limitations

The research was conducted with acknowledgment of the following limitations:

1. The selection of students was limited to students enrolled in computer science core courses at Towson University during the spring and fall 2007 semesters.
2. The courses identified for security integration were taught by four instructors; the

study was limited due to possible variations in teaching style and syllabi of the instructors.

3. This research used student pretests and posttests. Although it is assumed that students answered questions to the best of their ability, this study was limited due to the accuracy of this assessment.

2. Literature Review

Software security is the process of building software that is resistant to attack. This literature review begins with a general overview of software security and describes the ongoing security crisis, particularly in education, which currently employs a bottom-up approach that teaches students to program first and learn secure coding later. Recent literature advocates an integration approach, that interleaves security principles throughout the instruction Reports from SANS (SANS, 2007) indicate that most vulnerabilities can be traced to lack of security knowledge, declaration errors, failure to validate input, buffer overflow, and logic errors. This review will examine the body of literature that discusses software security best and worst practices, security integration and active learning in education, checklists, and risk management.

2.1 Software Security

Attacks on internet–connected systems have become so commonplace, that as of 2004, the CERT Coordination Center (CERT/CC) stopped reporting this number (CERT, 2007). An attack is a security threat only if there is vulnerability. While the number of reported vulnerabilities decreased slightly in 2007, the numbers remain ominously high.

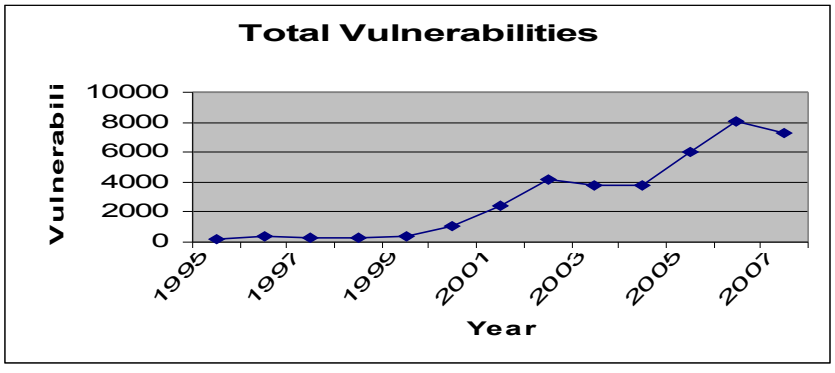

Figure 1: Total Vulnerabilities reported by CERT

The President's Information Technology Advisory Committee (PITAC) report

entitled "Cyber Security: A Crisis of Prioritization" estimates that current software development techniques produce 1 to 7 defects per thousand lines of code (2005). For a large system with millions of lines of codes, that translates to thousands of vulnerabilities.

Software security is a relatively new field which garnered a tremendous amount of publicity when Microsoft launched its Trustworthy Computing Initiative (TCI) precipitated by Bill Gates famous "security evangelist" memo (Microsoft, 2002). Gates committed the company to fundamentally change its mission and strategy in the key areas (referred to as the four pillars) of Security, Privacy, Reliability, and Business Practices. Microsoft's TCI is a long-term, collaborative effort to create and deliver secure, private, and reliable computing experiences for everyone. Key to Microsoft's TCI was Michael Howard, a software security expert and prolific author who advocates the Secure Development Lifecycle, which incorporated security at all phases of the traditional software development lifecycle. Howard cites increased interconnectivity and the "Wild, Wild Web" as placing new demands on software (Howard & Leblanc, 2002). He encourages programmers to write code as if it will execute in a hostile environment and to treat all bugs as exploitable and repair accordingly. He outlines an "SD^3" model: a system that is secure by design, by default, and in deployment. Howard calls for more education regarding secure design and secure coding, and more thorough testing.

Another software security pioneer, Gary McGraw, co-authored *Building Secure Software*, (Viega and McGraw, 2001), one of the first books to appear on the topic and emphasizes the idea of building security in, which they refer to as a white hat approach to security. In 2004, McGraw's second text, *Exploiting Software: How to Break Code*

(Hoglund & McGraw, 2004) describes the black hat approach, specifically how software attackers exploit vulnerabilities. With his third text, *Software Security: Building Security In*, McGraw (2006), "unifies the two sides of software security--attack and defense, exploiting and designing, breaking and building--into a coherent whole." McGraw attributes growing security problems to the "trinity of trouble--connectivity, extensibility, and complexity" and outlines a solution he refers to as the "three pillars of software security": applied risk management, software security best practices (called touchpoints), and knowledge. He advocates incorporating security engineering into all phases of systems development, designing for security, and risk analysis and testing. McGraw (2006) states that "software security is about building secure software: designing software to be secure, making sure that software is secure, and educating software developers, architects, and users about how to build secure things."

Anderson and Whitaker (2004) list five goals of computer security: confidentiality, integrity, availability, authenticity, accountability and describe the security triangle of attackers, assets, and software vulnerabilities.

2.1.1 Best Practices

There are numerous lists of touchpoints, best practices, mantras, etc. in software security. All can be traced back to the following original principles outlined by Saltzer and Schroeder (1975):

- **Economy of Mechanism**-The protection mechanism should be as simple as possible.
- **Fail-safe Defaults**-The protection mechanism should deny access by default, and grant access only when explicit permission exists.

- **Complete Mediation**-The protection mechanism should check every access to every object.
- **Open Design**-The protection mechanism should not depend on attackers being ignorant of its design to succeed but may be based on the attacker's ignorance of specific information such as passwords or cipher keys.
- **Separation of Privilege**-The protection mechanism should grant access based on more than one piece of information.
- **Least Privilege**-The protection mechanism should give a process the minimum privileges needed to perform its task.
- **Least Common Mechanism**-The protection mechanism used to access resources should not be shared.
- **Psychological Acceptability**-The protection mechanism should be (relatively) easy to use.

Howard and LeBlanc (2003) list "security principles to live by": Learn from Mistakes, Minimize Your Attack Surface, Employ Secure Defaults, Use Defense in Depth, Use Least Privilege, Backward Compatibility Will Always Give You Grief, Assume External Systems Are Insecure, Plan on Failure, Fail to a Secure Mode, Remember That Security Features != Secure Features, Never Depend on Security Through Obscurity Alone, Don't Mix Code and Data, and Fix Security Issues Correctly. In their excellent book, *Secure Coding Principles and Practice*s, Graf & van Wyck (2003) outline 30 principles of security architecture, including "engineer security in from day one" and "design with the enemy in mind." Secure design steps include assessing

risks and threats and a risk mitigation strategy. At the implementation level, they offer specific steps for handling data including: perform bounds checking, check configuration files, check command-line parameters, environment variables, set valid initial values for data, understand filename references and use them correctly, be wary of indirect file references, and pay special attention to the storage of sensitive information. McGraw (2006) defines three pillars of software security and seven touchpoints:

Three pillars	**Seven Touchpoints**
knowledge.	Code review
risk management	Architectural risk analysis
touchpoints or software security best practices	Penetration testing
	Risk-based security tests
	Abuse cases
	Security requirements
	Security operation.

Table 1: Software security: Best Practices.
McGraw (2006)

2.1.2 Worst Practices

Just as there are several lists of software security best practices, there are also lists of mistakes or "sins" that programmers commonly make.

Howard (2005) lists 19 deadly sins: 1. Buffer overruns 2. Format string problems 3. Integer overflows 4. SQL injection 5. Command injection 6. Failure to handle errors 7. Cross-site scripting 8.Failure to protect network traffic 9. Use of magic URLs and hidden forms 10. Improper use of SSL 11. Use of weak password-based systems 12. Failure to store and protect data securely 13. Information leakage 14.

Trusting network address resolution 15. Improper file access 16. Race conditions 17. Unauthenticated key exchange 18. Failure to use cryptographically strong random numbers 19. Poor usability. Thompson and Whitaker (2003) describes the "seven habits of highly insecure software" and McGraw (2006) describes seven (plus one) kingdoms of software security errors:

Seven habits of highly insecure software (Thompson & Whitaker, 2003)	**Seven (plus one) kingdoms of software security errors: (McGraw (2006)**
Poorly Constrained Input	Input Validation and Representation
Temporary Files	API Abuse
Securing Only the Most Common Access Route	Security Features
Insecure Defaults	Time and State
Trust of the Registry and File System Data	Error Handling
Unconstrained Application Logic	Code Quality
Poor Security Checks with Respect to Time.	Encapsulation
	Environment.

Table 2: Software security: Worst Practices.

A recent report by the SANS Institute (2007) shows that the programming errors that were responsible for more than 85% of the most critical vulnerabilities were:

1) failure to properly validate input
2) buffer overflow
3) integer overflow

Buffer overflow, particularly infamous in C and C++ programs, is one of the

oldest and most prevalent vulnerabilities. Attacks uses memory manipulating operations to overflow a buffer resulting in the modification of an address to point to malicious or unexpected code. Some work has been done in introducing this concept to undergraduate education (Gerhart, 2003; Werner & Frank, 2006).

Increasingly, C and C++ are under attack for security issues. As a result, improved standards, compiler implementations and runtime analysis tools have been developed. In addition to the issues above, other specific secure coding recommendations from texts (Howard, 2003; Seacord, 2006; Viega & Messier, 2003) include:

- Avoid the use of c strings and if necessary make use of new safe string libraries such as strsafe.h.
- Use appropriate complier flags such as /GS in the Visual c++ .net environment.
- Robust initialization
- Beware of temporary files and any filenames that are input.
- Use unsigned ints, if possible, or SafeInt, include integer validation and range checking
- Watch for dangling pointers – increasingly used for attacks

2.1.3 Risk Management

Key to software security is risk management. Risk analysis is well defined, and begins with identifying and ranking threats and vulnerabilities. The risk management process is a full lifecycle activity that includes identifying, synthesizing, ranking, and

keeping track of risks throughout software development for both large and small projects (McGraw, 2006, Perrone et al 2005).

Microsoft's version of this process is called threat modeling (Howard, 2003; Meier et al, 2003; Swiderski & Snyder, 2004):

1. *Identify assets* that your systems must protect.
2. *Create an architecture overview* using diagrams and tables.
3. *Decompose the application* including the underlying network and host infrastructure design, to create a security profile for the application.
4. *Identify the threats* using the STRIDE model, an acronym for Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service, and Elevation of Privilege
5. *Document the threats* using a common threat template that defines a core set of attributes to capture for each threat.
6. *Rate the threats* to prioritize and address the most significant threats first. The rating process weighs the probability of the threat against damage that could result should an attack occur. Some threats may not warrant action when you compare the risk posed by the threat with the resulting mitigation costs.

The following statement appears on Microsoft's website (http://msdn2.microsoft.com/en-us/security/aa570412.aspx) “Microsoft Application Threat Modeling is a critical security activity, enabling effective application risk management during the SDLC and beyond. Application Threat Modeling is enforced as part of the Security Development Lifecycle for IT (SDL-IT) at Microsoft.” McGraw (2006) believes that Microsoft misuses the term threat modeling, but credits the ideas as

sound. The consensus - understanding threats is critical to building a secure system.

2.2 Software Security Education

Also critical to the security effort is the education of current and future software developers. Security experts emphasize the need to incorporate security into the undergraduate curriculum (Bishop & Frincke, 2005; Davis & Dark, 2003; Graff & van Wyck, 2003 ;Hoglund & McGraw, 2004; Howard & LeBlanc, 2003; Irvine et al. , 2003; McGraw, 2006, Perrone et al., 2005). Bishop and Frincke (2005) state, "The ability to write secure code should be a fundamental to a university computer science undergraduate as basic literacy. " It is the responsibility of universities to teach future computing professionals secure and robust coding and design principles often and early in the learning process. Parallel to the industry effort of integrating security into the software development lifecycle, security concepts need to be infused into all levels of the computing curriculum.

The importance of software security education is a recurring theme throughout the literature. Computer science graduates lack security training, software designers, testers, and engineers lack security skills. The summary of US National Cybersecurity Summit Subgroup of the "Security across the Software Development Lifecycle" task force lists awareness and education of current and future software developers as one of its key recommendations (Davis et al, 2004). Matt Bishop, an expert in software security states that "the ability to write secure code should be as fundamental to a university computer science undergraduate as basic literacy" (Bishop & Frincke 2005). They advocate a principles-based approach, "the emphasis in higher education is on designing security in from the the beginning rather than adding it afterward." Several

authors refer to incorporating principles of "robust programming", including defensive programming, hiding information, and assuming the impossible (Bishop & Frincke. 2004, Graff & van Wyck, 2003,Yasinsac et al, 2006). When students learn these principles and apply them, they will create programs that are more structured, more robust and consequently more secure. There is a fundamental lack of "security as threats" education available to student today and most security classes teach "security as crypto" (Howard,2004). Avoiding issues such as integer and buffer overflow doesn't lead to more secure software and may even result in cryptology algorithms with vulnerabilities. Education is a crucial part of the Microsoft TCI. Bill Gates states, "I think the most critically important part of delivering secure systems is raising awareness through security education" and education is a facet of the Security Development Lifecycle (Howard & Lipner, 2006). Many educators have joined forces with the SANS group to form a Secure Programming Skills Assessment initiative that includes over 300 corporations, government agencies, and colleges to develop four comprehensive examinations to assess secure programming skills.

Resources for security education curriculum development include the Colloquium for Information Systems Security Education (CISSE), the International Processing (IFIP) Working Group 11.8 on Information Security Education (WISE), and the Workshop on Education in Computer Security (WECS) (Frincke and Bishop, 2004). The National Security Agency (NSA) and the Department of Homeland Security (DHS) jointly support the National Centers of Academic Excellence in Information Assurance Education (CAEIA) program whose goal is to "reduce vulnerability in our national information infrastructure by promoting higher eduction in information assurance (IA),

and producing a growing number of professionals with IA expertise in various disciplines"(http://www.nsa.gov/ia/academia/caeiae.cfm). The CAEAI recognizes eligible colleges and universities as National Center of Academic Excellence.

2.3 Integrating Security

White and Nordstrom (1996), Irvine et al (1998) and Yang (2001) propose integrating security concepts within the information systems programs to promote security knowledge and encourage critical thinking. Vaughn (2000) outlines the lack of computer security training in computer science programs and recommends security integration in the following courses: Operating Systems, Software Engineering, Database, Networks, and Artificial Intelligence. He also recommends adding an Information Security Capstone course. A panel at SIGCSE (Mullins et al, 2002) advocates incorporating security-related content in computer courses including topics such as: 1) risks 2) spoofing 3) reconnaissance sw 4) encryption 5) op system vulnerabilities 6) DOS 7) virus & worm 8) remote monitoring 9) Trojan horses 10) secure email 11) firewalls 12) mobile code. The panel recommended security context assignments in upper and lower level courses as an intrinsically motivating way of increasing security knowledge. Davis and Dark (2003) recommend a holistic approach to teaching security and advocate threading important security concepts throughout the curriculum, for example discussing buffer overflows in conjunction with a discussion about stack frames. They assert that repetition of security skills will lead to internalization of best practice. A graduate project conducted at UMBC (Cree et al, 2004) suggested integrating and repeating defensive programming concepts into CS1, CS2, Data Structures, and Software Engineering.

Perrone et al (2005) present an extensive review of the current undergraduate

approaches to security: 1) single-course 2) track and 3) thread. While the single-course technique is popular, the amount of students reached is limited. The track or sequence of courses, is resource-intensive, occurs late in curriculum, and fails to reach all students in the majors. Conversely the thread technique, which integrates computer security across the core curriculum, bridges the gap, and can be applied across different kinds of institutions. They assert that security should be a "recurring theme" across undergraduate curriculum in Computer Science and Engineering. Repeated exposure will lead to reflection and assimilation and complement the effectiveness of security electives. The strengths of the thread include:incremental integration requires no drastic changes; individual faculty can develop materials and integrate gradually; no additional courses are required; and security is a unifying theme .

"Where Security Education is Lacking" (Pothamsetty, 2005) describes current efforts in education (teaching attack techniques, security technologies, etc.) as addressing the symptoms of security vulnerabilities as opposed to the symptoms of insecure software. "The time has come for the academic community to start working towards integrating security into the core learning objectives of Computer Science and Engineering so that every undergraduate that successfully completes our degree programs will have a level of understanding of security and its importance to the design and the development of information systems." He recommends embedding security concepts early and often across the undergraduate CS curriculum along with a secure software engineering course. He sees "small incremental changes in syllabi" as a way to unify the

students body of knowledge and recommends specific content including: secure programming in CS1 and CS2; memory protection and buffer overflow in Computer Organization; protection, reinforce, virtualization, access control, in Operating Systems; type safety, security models, virtual machine, mobile code, access control in Programming Languages; risk and privacy into an Ethics course.

Integrating basic information assurance knowledge and secure coding into general CS classes is also advocated by Harrison et al (2006). Insecure coding continues, current texts and instruction ignore security. “Although it cannot be denied that security has gained immense importance over the last several years, it is still relegated to special ‘Security courses’ and is not, well-integrated into the general computer science curriculum. Programming education remains unchanged, emphasis on making programs run, we are creating a generation of insecure coders who “code like their teachers, using insecure code without necessarily even realizing that it is insecure” fail to teach security in a timely manner – hence the buffer overflow problem.”

Conklin and Dietrich (2007) advocate an integration approach that mixes testability and security with “primitives” (basic coding concepts such as i/o, data and control structures) and echoing the traditional engineering approach which integrates risk analysis, failure mode effects analysis and reliability throughout. To address the problem of curriculum overload, they outline some example that can be employed in the core classes, such as good documentation, Try-Catch-Finally (error handling), input validation and buffer overflow management. He also believes that the higher level security issues will enforce Bloom's taxonomy (Bloom, 1956), which classifies learning into six levels: knowledge, comprehension, application, analysis, synthesis, and evaluation; and move

learning from comprehension to application.

Despite the increasing amount of literature that advocates integrating security into the curriculum, there are a only a few specific integration examples or ideas: security integration in a Software Engineering class (Shumba et al, 2006) and CS1 security topics (Azadegan & O'Leary,2006; Null,2004; Walden, 2006). The proposed threaded model is not widely implemented. Currently, security integration is promoted but not widely practiced.

2.4 Checklists

Checklist are used in industry, most notable aviation, to enforce safety procedures. Checklists are also increasingly used in software development. Gilliam et al (2003) advocate Software Security Checklists (SSC) for use as walk-throughs at each phase of the software lifecycle and to identify input points in the source code and argues that a checklist will lead to more secure software. Risk checklists have also been shown to effective at helping software practitioners identify risks and "heighten sensitivity to risk" (Keil et al, 2006).

Checklist have also been recommended for use in education. Bishop (2006) describes the role of checklists as an enumerative reminder list, a requirements list, and an evaluative tool. "The best checklists are derived from principles, and their items develop logically through the derivation of principles, methodology, and application to a particular domain, guided by experience of practitioners. This allows one to justify the checklist rigorously and to see how the principles strengthen the practice, assurance at its

best." He cautions, "Used properly, a checklist can enhance a students' education; used improperly, that same checklist can hinder the student's progress, as well as fail to achieve the goals of that student's education." Bishop and Frincke (2005) address the role of checklists in assurance by paralleling the use of pilot checklists which require student pilots to have a strong foundation of airline safety principles. Similarly, checklists can help with secure coding by supplementing the educational process. Graff and Van Wyck (2003) advocate the use of checklists because of psychological difficulties of designing secure code. They also discuss the concept of re-using a checklist as an evaluative tool or scorecard. This study builds upon, extends, and puts into practice the checklist concepts introduced by Bishop and Frincke and Graff and Van Wyck.

2.5 Additional Security Topics

Additional areas for integration include programming languages and databases. In the area of programming languages, Skalka (2005) discusses current research in "language safety." Language safety is tied to systems security as attackers typically exploit vulnerabilities in programs related to language design and implementation, such as buffer overflow. Sklaka (2005) cites Java and C# as safe modern languages (have a strong typing or type safety) and equipped with sophisticated security models, such as access control policies. Type safety is a property of some programming languages (not C) that involve the use of a type system to prevent certain erroneous or undesirable program behavior either statically, at compile time, or dynamically, at runtime (Pierce, 2002).

In the area of databases, the risk of SQL injections is increasing because of

automated tools. SQL injection is a security exploit in which the attacker adds SQL code to a Web form input box to gain access to resources or make changes to data. To mitigate injection risks: validate the input by only accepting valid input, use parameterized queries and stored procedures, and employ the principle of least-privilege when connecting to the database server (Howard, 2003).

2.6 Active Learning

Active learning is a pedagogical approach that provides students opportunities to relate interactively with subject matter, encouraging them to generate rather than to simply receive knowledge. Active learning fosters the high-order thinking tasks of analysis, synthesis, and evaluation. A "learning by doing" approach encourages students to engage with the material through a variety of "active" activities including, for example, class and small-group discussions, reflective exercises, think-pair-share activities, case studies, collaborative learning. Most research shows that active learning techniques yield positive learning gains: students learn more, they retain the information longer, they can apply learned material in more contexts, and the environment of the classroom is more enjoyable (McConnell, 1996)

Active learning is not a new idea. The "buzz word" of the 1980's has re-emerged. The current view is that active learning is most effective when used to enhance the traditional learning environment. Many teachers recognize that the traditional lecture format, though necessary, requires supplementary activities to reinforce the subject matter. Formally or informally, many teachers employ "learning by doing" techniques in their classrooms, such as discussion questions, demonstrations, case studies, and group

work, to stimulate interest and engage student in the subject matter.

Active learning is particularly appropriate for computer science students. Research shows that CS students tend be active and visual learners and learn best in an active learning environment (Briggs, 2005). However, most computer science classes are conducted in a traditional lecture format supplemented with laboratory assignments. Computer science is a challenging curriculum; drop-out rates are high, particularly in the first courses. Recent studies have shown that employing active learning techniques in CS classes increases learning and improves the environment (Briggs, 2005, McConnell, 1996).

2.7 Summary

In this chapter, we describe the software security crisis, including various experts lists of best and worst practices. We outline the current bottom-up educational approach, which currently teaches students to program first and learn secure coding later and review literature that introduces an integration approach that interleaves security principles throughout the instruction. Using the ideas discussed by White and Nordstrom (1996), Irvine et al (1998), Yang (2001), Davis and Dark (2003), Perrone et al (2005), Pothamsetty(2005), Harrison et al (2006) and Conklin and Dietrich (2007), we have developed, implemented and evaluated a security integration curriculum, where the use of security checklists (Bishop and Frincke, 2005, Graff and Van Wyck, 2003) is a key component. Our model begins where the literature leaves off, the details are outlined in the following chapters.

3. Methods

This research included the development of security modules, injection of the modules into the undergraduate computing curriculum, and pre and post-assessment. This chapter outlines the research methods used and describes the sample, the security assessment, the security lab modules, and the data collection and analysis.

3.1 Sample

Towson University is a mid-size institution with approximately 16,000 undergraduates. The Computer and Information Sciences Department comprises two majors, Computer Science (CS) and Computer Information Systems (CIS). Currently, there are approximately 150 students in each major.

The CS and CIS programs require the core courses CS1 and CS2. CS1 requires prior programming experience, which students can satisfy by taking CS0. The courses are 15 week four-credit classes and are described below:

CS0: *COSC175 - General Computer Science* logic course taught in pseudocode

CS1 : *COSC236 - Introduction to Computer Science I* introduces imperative programming using C++

CS2: *COSC175 – Introduction to Computer Science II* covers object oriented concepts using C++

Each of the courses include a lecture and lab component. Syllabi for each course are included in Appendix A. This research study was conducted spring 2007 and fall 2007. In spring 2007, we piloted security integration across all three sections of CS0

and two of five sections in CS1. In fall 2007, we repeated security integration in one of three section of CS0, three of four sections of CS1, and one of four sections of CS2. We conducted security pretests and posttests across all sections and compared the results of the integration effort.

Additionally, we began development and preliminary integration of modules in Systems Analysis and Design, Programming Languages: Design and Implementation, and Database Management (COSC and CIS). These courses are described below:

Systems Analysis and Design (CIS179): covers the evolution of data processing systems, including analyses of information flow, system specifications, interface design and system implementation, required for CIS majors

Programming Languages: Design and Implementation (COSC455): discusses principles of various programming language paradigms, required for COSC majors

Database Management Systems (COSC457, CIS458): topics include data models and sublanguages, implementation and use of a database management system, comparison of widely used DBMS packages, required for COSC and CIS majors

3.2 Security Assessment

To evaluate the effectiveness of our integration effort, a security test was administered, both pre- and post-, to all students in CS0, CS1, and CS2. The purpose of the test was to assess general and specific security awareness and knowledge. The test was multiple-choice, consisting of 6 demographic questions and 20 awareness/knowledge questions. Tests were administered on-line, during the first and last lab period. The test was anonymous, voluntary, and was administered on-line. Completing the test had no

effect on student's grades or status as a student athlete.

In spring, 150 students took the pretest and 109 students took the posttest. In the fall, 242 students took the pretest and 166 took the posttest.

3.3 Security Modules

The primary vehicle for our security integration was a series of laboratory modules. CS and CIS curriculum and instructors are already over-extended; the laboratory is one area with some flexibility. Additionally, since developing meaningful lab assignments is both challenging and time-consuming, we intend the security lab modules to be useful for students and instructors.

The structure of our laboratory modules borrows from labs in the traditional sciences such as biology, chemistry, and physics. The format is as follows:

1. Background
2. Problem: Security-related lab
3. Security Checklists
4. Analysis

Complete lab modules appear in Chapter 4.

3.3.1 Touchpoints

Our model highlights the following specific security-related topics which have been identified by the SANS Institute as the three programming issues accounting for more than 85% of all security vulnerabilities reported in 2006 (SANS, 2007):

Integer Overflow- Integer overflow occurs when a number exceeds the largest possible value that fits in the allocated space for a variable. Integer overflows yield

unexpected behavior, which may be ignored or cause an abort. If the overflow is ignored, the program may crash or store erroneous data. Attackers can exploit integers used as array indices, loop counters, or lengths, to create buffer overflows and execute malicious code. Integer overflows are difficult to detect and attacks of this type are increasing (Seacord, 2006). While avoiding signed numbers simplifies the secure coding process (Seacord, 2006), in our checklists, we emphasize strong and appropriate typing. The goal is to instill secure coding habits for future programmers.

Buffer Overflow - Considered the "nuclear bomb" of the software industry (Hoglund & McGraw, 2004), the buffer overflow is one of the most persistent security vulnerabilities and frequently used attacks. Attackers can exploit buffer overflows to modify code and data structures and change function pointers . The infamous buffer overflow gained public notoriety in 1988, yet still accounts for up to 50% of all software vulnerabilities (Viega & McGraw, 2002). While any discussion of C and C++ includes warnings about buffer overflow, in our model, the security implications of buffer overflow are demonstrated and reinforced with labs and checklists following a discussion of program run-time environment and memory layout.

Input validation – Many security problems result when programs fail to properly sanitize the input data.. Traditionally, programmers have written code to satisfy requirements with an emphasis on making things possible. Secure coding requires a shift of focus towards making potentially dangerous actions impossible. For example, code that attempts to reject all incorrect data, a potentially infinite set, is both inefficient and insecure. Secure code accepts only valid data and rejects anything else. In other words, deny by default. When teaching students proper input validation, the following security

mantras are emphasized:

- basic understanding of the hostile environment
- assume the impossible
- all input is evil
- graceful degradation
- deny by default

Other topics include discussion of the SDL, which incorporates security across the software development lifecycle, the concept of software vulnerabilities and threats, malware, and using secure passwords and pins.

3.3.2 Background

The background section is comprised of three sections: 1) definitions, 2) real-life examples, and 3) research questions. Definitions include brief descriptions of the terms being used in the lab. Real-life examples include links to pertinent articles offering actual occurrences of security vulnerabilities. Research questions ask students to define other security concepts or provide thoughtful answers or opinions pertaining to the article they have read. The purpose of this section is to provide background information necessary for the security lab and to start students thinking about the security touchpoints introduced in the module.

3.3.3 Security-Related Lab

The traditional CS core concepts, or primitives, such as loops, decisions, etc., are a required component of the curriculum. In our model, the labs used to reinforce these concepts will now include a related security topic. For example, when data types are introduced in CS0, the lab will demonstrate integer overflow. Or, a security topic, such as

PIN generation, may be used to reinforce a primitive such as loops. The table below shows how security topics and core concepts are included in the security modules.

Security Topic	**CS0 concept**	**CS1 concept**	**CS2 concept**
SDL	Software lifecycle		
Integer overflow 1 Integer overflow 2	Data operators	Data, operators loops	
Buffer overflow	arrays	arrays	arrays
Input validation	decisions	decisions	
files		vulnerabilities	vulnerabilities
Password, pins		Strings, loops	
Dangling pointers			Dynamic data

Table 3: Security Lab: Mapping security topics to core concepts

3.3.4 Security Checklist

To complement the security labs, we have developed a series of checklists. Checklists are used in many applications to reduce the likelihood of human error, most notably in aviation, where pre-flight checklists are a key method in improving airline safety, and, increasingly, in software assurance. Security education expert Bishop advocates the use of principle-based checklists in industry (Bishop, 2006) and in education (Bishop & Frincke, 2005). A well-developed checklist serves as a reminder list and helps to ensure consistency and completeness. Security checklists reduce the likelihood of omitting a key security feature and provide a quantifiable list of criteria (Graff & van Wyck, K. 2003) . Designing fully secure code is psychologically difficulty (Graff & van Wyck, K. 2003): “why would anyone do that?” or “you want me to break my own code?” Checklists can help to find potential vulnerabilities. Ultimately, checklists reinforce security principles and help the student internalize key security

concepts.

The basis for our checklists are well-established security mantras that guide software security (Taylor & Azadegan, 2007) . Of course, security checklists cannot find all security flaws; and there is no such thing as completely secure code This project begins with a series of checklists designed to target a few of the most common vulnerabilities that account for most security issues. Effective use of security checklists requires training, repetition, and feedback.

The checklist have been developed and introduced in an iterative manner. The initial checklist presented in CS0 , see Figure 2, addresses only integer overflow. As new topics are introduced, additional items are added to the checklist. This allows students to gradually adapt to using checklists with their programs. The same technique is used in CS1 and CS2, culminating in the final checklist, shown in Figure 9.

3.3.5 Analytical Questions

The final component of the security modules includes discussion questions that require students to analyze and summarize the results of the labs. Modeled after traditional scientific labs, which employ Bloom's taxonomy approach to learning (Bloom, 1956), the questions promote critical thinking and reflection. Technology education has inadequately addressed synthetic and analytical thinking (Levin & Lieberman, 2000; Taylor & Azadegan, 2007) and the goal for most labs is to "get them running." The inclusion of discussion and feedback questions requires students to consider and analyze the process, the results, and, in this case, the security implications.

Additionally, students' answers to these questions provide valuable and

immediate feedback to the instructor. Typically, an instructor determines the class level of mastery through exams, after discussion of a topic is completed. Including discussion questions with each lab allows the instructor to assess how well a concept has been assimilated before moving on to the next topic.

3.4 Instrument

In this study, an assessment test was created by the author containing 20 content questions and 6 demographic questions. The test was administered online at the beginning and the end of each semester across all security-integrated and control sections. The instrument tested for overall reliability with a value of .74 (Cronbach alpha). A copy of the test can be found in Appendix B.

3.5 Limitations and Assumptions

This research was conducted with the acknowledgment of the following limitations:

1. The selection of subjects was limited to 392 students in three sequential major computer science courses at Towson University during Spring 2007 and Fall 2008. The sample was a sample of convenience and introduced bias. Results of this study are not generalizable beyond this sample of 392 eligible students in these courses.
2. The courses included different teachers each semester and although materials were distributed uniformly, the study was limited due to possible variances in teaching style of the different instructors.

3. This report used student questionnaires. Although it is assumed that students answered questions truthfully and honestly, this study was limited due to the individual differences in student self-assessment.
4. The questionnaire could only be administered to students who actually attended class when the test was given; there is no data for students who did not attend.

3.6 Institutional Review Board

Study approval by Towson University's Institutional Review Board (IRB) for Research involving the use of Human Participants was granted under Exemption Number 07-1X67. The research was exempt from general Human Participants requirement according to 45 CFR 46.101(b)(2). As noted earlier, participation in the study was voluntary, anonymity of the participant was insured, and the participant was fully informed of the research project. A copy of the IRB approval can be found in Appendix A.

3.7 Data Collection and Analysis

Data was collected using a database driven World Wide Web instrument and was entered into a statistical package (SPSS) for later analysis. The format of the experiment was the Pretest-Posttest Control Group Design. For each course, CS0, CS1, and CS2, some sections were designated as security sections and received security-integration modules. The remaining sections served as controls and received no security integration. Comparisons were then made between: posttest scores of control and security-integrated, gain scores of control and security-integrated, and pretest scores of control and security-integrated.

3.8 Summary

This study evaluated the effectiveness of security modules incorporated into several sections across three computer science introductory courses over the course of two semesters. A new instrument was developed to assess and compare learning across all sections. A pilot study was conducted and the instrument was tested for reliability and usability. Reliability testing was also performed in the actual study and confirmed results of the pilot study. In the actual study, the instrument tested for overall reliability with a value of .74. Study approval by Towson University's Institutional Review Board (IRB) for Research Involving the Use of Human Participant was granted under Exception Number 07-1X67 on 5/1/07 (Appendix A).

4. Security Modules

4.1 CS0 Lab 1

A. Background

Primitives: Introduction to compiler, syntax error vs complier error
Security: SDLC

B. Problem

```
// Software Development Lifecycle
// Written by: _____________________
#include <iostream>
using namespace std;
int main()
{
    cout << "There are 5 steps in the software development lifecycle (SDLC)." << endl;
    cout << " 1. Define the problem." << endl
    cout << " 2. Plan the solution." << endl;
    cout << " 3. Code the solution." << endl;
    cout << " 4. Test and debug." << endl;
    cout << " 5. Maintain and Document. " << endl;
    cout << " 6. Add Security Features." << endl;
    return 0;
}
```

1. **Create a project:** Print out the Visual C++ Handout. Follow step 1-5.
2. **Edit:** Step 6: Type in the above program. Include your name.
3. **Compile:** Step 8. Correct the syntax error.. Correct it and compile again.
4. **Run:** Follow steps 9 and 10. Examine the output carefully.
5. **Turn in:** program and answers to questions below.

C. Analysis

1. What is the role of the compiler?
2. What is a syntax error? What was the syntax error in the above program? What happens when you have a syntax error in your program?
3. What is a logic error or bug? What was the logic error in the above program?
4. In one sentence, summarize the difference between syntax errors and logic errors.
5. Where do you think security should fit into the SDLC?

4.2 CS0 Data Lab

A. Background

Primitives: Data types
Security: checklists, integer overflow

Integer overflow is the result of trying to place into computer memory a value that is too large for the allocated space. Integer overflow can lead to erratic system behavior, buffer overflows, and the execution of arbitrary code by an attacker.

Real world example: On 12/25/2004, Comair halted operations and grounded 1,100 flights after a crash of its flight crew scheduling software. The number of monthly changes was stored in a 16-bit integer. A series of storms in early December caused the crew reassignments to exceed this number.
http://www.cincypost.com/2004/12/28/comp12-28-2004.html

1. Describe the integer data type.
2. Describe the float data type.
3. Give an example of a variable that would use the integer type.
4. Give an example of a variable that would use the float type.

B. Problem

```
#include <iostream>
#include <limits>
using namespace std;
int main()
{
   // variable declarations
   float x;
   int   i;
   char  ch;

   //integer
   cout << "For this compiler: " << endl;
   cout << "largest integer is: " << INT_MAX << endl;
   cout << "smallest integer is: " << INT_MIN << endl;
   cout << "size of an integer is "<< sizeof(int) << " bytes." << endl;
   cout << "Enter an integer value " << endl;
   cin >> i;

   // float
   cout << endl << "size of a float is "<< sizeof(float) << " bytes." << endl;
   cout << "Enter a float value " << endl;
   cin >> x;

   //character
   cout  << endl << "size of a char is "<< sizeof(char) << " bytes." << endl;
```

```
    cout << "Enter a character value " << endl;
    cin >> ch;

    //output
    cout << endl << "You entered the following values: " << endl;
    cout << "integer:   " << i << endl << "float:      " << x << endl;
    cout << "character: " << ch << endl;

    //integer overflow
    i = INT_MAX + 1;
    cout << "integer overflow: i = " << i;
    return 0;
}
```

1. *Create* a new project. *Type* in the above program. *Compile* and *run*.
2. *Print* out the *program* and *output* before proceeding.
3. From the above program, list each variable and the data type.

Variable	*Data Type*

C. Checklists

Checklists are used in many industries including aviation and software for safety and error checking. This semester we will use checklists to confirm the security of our code. Complete the following checklist for the program you just ran.

Security Checklist
For each integer:
• Mark with V any assignment operation (check all =) that could potentially overflow (multiplication is particularly risky)
• Mark with V any input (check all cin >>)

Figure 2: Security Checklist – CS0.1

D. Analysis

1. What is the largest integer value you can enter?
2. What happens when you exceed the largest integer value?
3. Why do you think that is?

Turn in program (marked after completing checklist), output, and questions.

4.3 CS0 Ops Lab

A. Background

Primitives: Operators
Security: vulnerability, integer overflow, divide by 0

Vulnerability-A weakness in a computing system that can result in harm to the system or its operations, especially when this weakness is exploited by a hostile person or organization or when it is present in conjunction with particular events or circumstances.

1. Explain the following statement: Most security violations are the results of software vulnerabilities.
2. Why is security a major concern in the computer industry today?
3. Explain the difference between security software and software security.

B. Problem

```
#include <iostream >
using namespace std;
int main()
{
        int num1,num2,sum, diff, remainder, prod;
        float quot;

        cout << "Enter two integer values" << endl;
        cin >> num1 >> num2;
        sum = num1 + num2;
        cout << num1 << " + " << num2 << " is " << sum << endl;
        diff = num1 - num2;
        cout << num1 << " - " << num2 << " is " << diff << endl;
        prod = num1 * num2;
        cout << num1 << " * " << num2 << " is " << prod << endl;
        quot = (float) num1 / num2;
        cout << num1 << " / " << num2 << " is " << quot << endl;
        return 0;
}
```

1. *Create* a new project. *Type* in the above program. *Compile* and *Run* with different values.
2. What happens when you enter very large numbers for num1 and num2?
3. What happens when num2 is 0? Why?
4. What does this tell you about using division or modulus in your programs?
5. *Add* the following lines to correct the divide by zero problem. Compile and run with different values to ensure it works correctly.

```
if (num2 == 0)
   cout << “cannot divide by zero!” << endl;
else
{  ....}
```

6. *Print* out the *program* and *output* before proceeding.

C. Checklists

Complete the following checklist for the program you just ran.

Security Checklist
For each integer:
● Mark with V any assignment operation (check all =) that could potentially overflow (multiplication is particularly risky)
● Mark with V any input (check all cin >>)
● Mark with V division that is not validated (divide by zero could occur)

Figure 3: Security Checklist – CS0.2

D. Analysis

1. Which operation is particularly vulnerable to integer overflow? Why?
2. When do you need to check for 0?

Turn in program (marked after completing checklist), output, and questions.

4.4 CS0 Selection Lab

A. Background

Primitives: If/if-else, switch
Security: input validation, vulnerability, integer overflow, divide by 0
Security Mantra: All Input is Evil!

- most (if not all) active attacks are the result of some type of input from an attacker
- Secure programming - ensure that inputs from bad people do not do bad things
- Defensive technique -> Input validation

Input validation (partial list):

- **Range check** - numbers checked to ensure they are within a range of possible values, e.g., the month of a person's date of birth should lie between 1 and 12.
- **Reasonable check:** values are checked for their reasonableness, e.g. (age > 16) && (age < 90)
- **Menu option check:** items selected from a menu or other sets of choices are checked to ensure they are available options
- **Divide by Zero:** variables are checked for values that might cause problems such as division by zero

1. Explain the following statement: Most security violations are the results of software vulnerabilities.
2. Why is security a major concern in the computer industry today?
3. Explain the difference between security software and software security.

B. Problem

```
#include <iostream>
using namespace std;
int main()
{
    int testScore;

    cout << "Enter test score" << endl;
    cin >> testScore;

    //input validation
    if ((testScore < 0) || (testScore > 100))
        cout << "Invalid score" << endl;
    else
    {
        if (testScore >= 90)
           cout << "Your grade is A" << endl;
        else if (testScore >= 80)
            cout << "Your grade is A" << endl;;
        else if (testScore >= 70)
            cout << "Your grade is A" << endl;
        else if (testScore >= 60)
            cout << "Your grade is A" << endl;
        else
            cout << "Your grade is A" << endl;
    }
    return 0;
}
```

1. Compile and run the above code.

C. Checklists

Complete the following checklist for the program you just ran.

Security Checklist
For each integer:
● Mark with V any assignment operation (check all =) that could potentially overflow (multiplication is particularly risky)
● Mark with V any input (check all cin >>)
● Mark with V division that is not validated (divide by zero could occur)
2. For all input:
● Mark with V any input that is not immediately validated (check all cin >>)

Figure 4: Security Checklist – CS0.3

D. Analysis

1. What type of input validation (see prev. page) is being used above?
2. Turn in program (marked after completing checklist), output, and questions.

4.5 CS0 Array Lab

A. Background

Primitive: arrays
Security: Buffer Overflow

A **buffer overflow** occurs when data is written beyond the boundaries of a fixed length buffer overwriting adjacent memory locations which may include other buffers, variables and program flow data. C++ is particularly vulnerable to buffer overflow.
Risk: Writing outside the bounds of a block of allocated memory can corrupt data, crash the program, or cause the execution of malicious code.
WARNING: over 80% of security problems result from buffer overflows!
Real world Example: The earliest known exploitation of a buffer overflow was the Morris worm in 1988. In 2001, the Code Red worm exploited a buffer overflow in Microsoft's Internet Information Services and in 2003 the SQLSlammer worm compromised machines running Microsoft SQL Server 2000.

B. Problem

```
#include <iostream>
using namespace std;
const int NUM_TESTS = 5;
int main()
{

    int  sum = 0;
     int tests[NUM_TESTS]; // array declaration
    float avg;

    //input  test scores
    cout << " Enter " << NUM_TESTS << " test scores: " << endl;
    for (int i = 0; i < NUM_TESTS; i++)
    {
        cout << "Enter Test " << i + 1 << ": ";
        cin >> tests[i];
    }

   // print out test scores
    for (int i = 0; i < ??; i++)
        cout << “test[“ << i << “] = “ << test[i] << endl;

     // average tests
    for (int i = 0; i < NUM_TESTS; i++)
    {
        sum = sum + tests[i];
    }
    avg = (float)sum/NUM_TESTS;
    cout << "The average is " << avg << endl;
```

```
    return 0;
}
```

Type in the above program. Fill in the appropriate value for ??. Compile and Run with this value.

1. Replace the ?? with 20 this time. What happened? Why?
2. Change the 20 to the correct value.
3. Draw a picture showing what the array looks like in memory.

C. Checklist

Complete the checklist below for the program you just ran.

Security Checklist
1. For each integer:
• Mark with V any assignment operation (check all =) that could potentially overflow (multiplication is particularly risky)
• Mark with V any input (check all cin >>)
• Mark with V division that is not validated (divide by zero could occur)
2. For all input:
• Mark with V any input that is not immediately validated (check all cin >>)
3. For each array reference:
• Mark with V any assignment that could result in an overflow. EXA: arr[i] = 0; // could i >= ARR_SIZE?
• Mark with V any loop that has a variable boundary EXA: for (i = 0; i <= riskyInt; i++) array[i] = 0;// is riskyInt < ARR_SIZE?
• Mark with V any unvalidated indices that are input EXA: cin >> i; array[i] = x; // is i < ARR_SIZE?

Figure 5: Security Checklist – CS0.4

D. **Analysis:**

1. Do you understand what a buffer overflow is?
2. How could a buffer overflow occur in your program?
3. What happened when a buffer overflow occurs?
4. How can you prevent a buffer overflow from occurring?

Turn in:the final program (marked after completing checklist),output, and questions.

4.6 CS1 Lab 1

A. Background

Primitives: Data types, i/o
Security: vulnerability, introduction to integer overflow
In computer security, the word **vulnerability** refers to a weakness in a system allowing an attacker to violate the integrity, confidentiality, access control, availability, consistency or audit mechanisms of the system or the data and applications it hosts. The key to preventing integer vulnerabilities is to understand integer behavior in digital systems.

Real world example: On 12/25/2004, Comair halted operations and grounded 1,100 flights after a crash of its flight crew scheduling software. The number of monthly changes was stored in a 16-bit integer. A series of storms in early December caused the crew reassignments to exceed this number.
http://www.cincypost.com/2004/12/28/comp12-28-2004.html

B. Problem

For each problem below, turn in the program and sample output:

```
// DEMO:  seclab1.cpp
#include <iostream>
#include <iomanip>
#include <climits> // this header includes all the constants used in this program
using namespace std;
void ShowTypeInfo();
void ShowOverflow();
int main()
{

    ShowTypeInfo();
    // ShowOverflow();
    return 0;
}
//*********************************************************
// This procedure shows the size and maximum and minimum values for various c types
void ShowTypeInfo()
{
    cout << "For this compiler: " << endl;
    cout << "int is " << sizeof(int) << " bytes" << endl;
    cout << "Min is " << INT_MIN << endl;
    cout << "Max is " << INT_MAX << endl << endl;
    cout << "short int is " << sizeof(short int) << " bytes "<< endl;
    cout << "Min is " << SHRT_MIN  << endl;
    cout << "Max is " << SHRT_MAX << endl << endl;
    cout << "long int is " << sizeof(long int) << " bytes"<< endl;
    cout << "Min is " << LONG_MIN  << endl;
```

```
    cout << "Max is " << LONG_MAX << endl << endl;
    cout << "unsigned short int is " << sizeof(unsigned short int) << " bytes" << endl;
    cout << "Min is " << 0  << endl;
    cout << "Max is " << USHRT_MAX << endl << endl;
    cout << "unsigned long int is " << sizeof(unsigned long int) << " bytes" << endl;
    cout << "Min is " << 0  << endl;
    cout << "Max is " << ULONG_MAX << endl << endl;

}
//*********************************************************
// This procedure demonstrates overflow
void ShowOverflow()
{
    int i;
    unsigned int j;

    i = INT_MAX + 1;
    cout << "overflowed int " << i << endl;

    j = UINT_MAX + 1;
    cout << "overflowed unsigned int " << j << endl;

}
```

1. Run the following program and print the program and output.
2. For this compiler is *int* the same as *short int* or *long int*? Consequently, the variable declaration
   int x; is the same as _____
3. Why is the maximum value of an *unsigned int* larger than the maximum value for an *int*?
4. When should you use *unsigned int* vs *(signed) int*?
5. For each of the following give the appropriate declaration:
   - students at Towson, total: 18,921 (ugrad: 15,374 grad: 3,547)
   - Population of Baltimore 635,815
   - Population of Maryland 5.6 million
   - the world population (6.5 billion).
6. What is an overflow?
7. Remove the comment from the call to ShowOverflow, compile, and run.What happens when a *(signed) int* variable overflows?
8. List ways this could occur in your code.
9. What happens when an unsigned into variable overflows? (This is called wraparound)
10. Write a program that shows the population growth (use a short int) for Towson University assuming 10% increase each year. Show the population after year 1, year 2, ..., do not use a loop.

C. Analysis

1. How many years until overflow occurs?

4.7 CS1 Selection Lab

A. Background

Primitives: Selection,Input validation,If/if-else, switch
Security: vulnerability, integer overflow, divide by 0

Security Mantra: All Input is Evil!

- most (if not all) active attacks are the result of some type of input from an attacker
- Secure programming - ensure that inputs from bad people do not do bad things
- Defensive technique -> Input validation

Input validation (partial list):

- **Range check** - numbers checked to ensure they are within a range of possible values, e.g., birth date month should be between 1 and 12.
- **Reasonable check:** values are checked for their reasonableness, e.g. (age > 16) && (age < 90)
- **Menu option check:** items selected from a menu or other sets of choices are checked to ensure they are available options
- **Divide by Zero:** variables are checked for values that might cause problems such as division by zero
- **Format or picture check** -Checks that the data is in a specified format (template), e.g., dates have to be in the format DD/MM/YYYY.
- **Presence check** -Checks that important data are actually present and have not been missed, e.g., customers may be required to have their telephone numbers listed.
- **Check digits** -Used for numerical data. An extra digit is added to a number which is calculated from the digits. The computer checks this calculation when data are entered, e.g., The ISBN for a book. The last digit is a check digit calculated using a modulus 11 method.

1. Explain: Most security violations are the results of software vulnerabilities.
2. Why is security a major concern in the computer industry today?
3. Explain the difference between security software and software security.

B. Problem

1. Write a program that asks the user to enter a number within the range of 1 to 10. Use a switch statement to display the Roman numeral version of that number. *Input validation*: Do not accept a number less than 1 or greater than 10.
2. **Geometry Calculator:**Write a program that displays the following menu:

Geometry Calculator
1. Calculate the Area of a Circle
2. Calculate the Area of a Rectangle
3. Calculate the Area of a Triangle
4. Quit

Enter your choice (1-4):
Include *Input Validation:* Display an error message if the user enters a number outside the range of 1 through 4 when selecting an item from the menu. Do not accept negative values for the circle's radius, the rectangles length or width, or the triangle's base or height.

C. Checklists

Security Checklist
1. For each integer:
• Mark with V any assignment operation (check all =) that could potentially overflow (multiplication is particularly risky)
• Mark with V any input (check all cin >>)
• Mark with V division that is not validated (divide by zero could occur)
2. For all input:
• Mark with V any input that is not immediately validated (check all cin >>)

Figure 6: Security Checklist – CS1.1

Complete the preceding checklist for the program below.

```
#include <iostream>
using namespace std;
int main()
{
    int num1, num2;
    int sum, diff, prod;
    float quot;

     cout << "Enter two integer values" << endl;
     cin >> num1 >> num2;
     sum = num1 + num2;
     diff = num1 - num2;
     prod = num1 * num2;
     quot = (float) num1 / num2;
     cout << num1 << " + " << num2 << " is " << sum << endl;
     cout << num1 << " - " << num2 << " is " << diff << endl;
     cout << num1 << " * " << num2 << " is " << prod << endl;
     cout << num1 << " / " << num2 << " is " << quot << endl;
     return 0;
}
```

D. Analysis

1. What is an integer overflow?
2. How could an integer overflow occur in your program?
3. What happens when an integer overflow occurs?
4. Why is multiplication particularly risk?
5. How can you prevent an integer overflow from occurring?

Turn in programs (marked after completing checklist), questions, and output.

4.8 CS1 Loops Lab

A. Background

Primitives: while, for
Security: input validation, password
The while loop can be used for input validation (often in a function as we'll see later). Example:

```
cout << " Enter a number in the range 1 - 100: ";   //1
cin >> number;                                      //1
while (( number < 1) || (number > 100))             //2
{
   cout << "Error:Enter a number in the range 1 - 100: ";//3
   cin >> number                                        //4
}
```

B. Problem

1. Write a program that includes the following **while** loops and **LABEL EACH LINE OF THE LOOP WITH A COMMENT AND 1-2-3 OR 4**
   a) a while loop that repeatedly finds the circumference and area of a circle, given its radius (float). The loop will stop executing when a 0 or negative value is entered. Use a constant for PI (3.14159265). Be sure and echo the input and display all output to 3 decimal places.
   b) a second while loop that works as above, but instead asks the user if they want to enter radius (Y/N). Reject invalid values for radius.
   c) Write a while loop that repeats until a reasonable age for a student is entered (you decide what is reasonable). Use constants.
   d) Write a loop that asks for a password until a valid password is entered

C. Checklists

Complete the checklist in *Figure 6* for each program above.

D. Analysis

1. How could you fix (mitigate) the vulnerabilities above?
2. Which fix is easy and which is not?
3. Discuss the risk in not fixing each of the vulnerabilities.

Turn in program(marked after completing checklist), output, and questions.

4.9 CS1 Loops Lab 2/Files

A. Background

Primitives: Files, loops
Security: malware, virus, worm, hacker, black-hat, white-hat, vulnerabilit
Real life example:In March 2004, the Netsky-D worm spread via email as a PIF attachment under a variety of subject names (including Re: Approved, Re: Details, Re: Document, Re: Your letter). In addition to making registry-key changes, Netsky-D activated PC speakers into making a constant beeping noise.

1. Give good definitions for the following terms:
   a) Malware
   b) Virus
   c) Worm

In computer security, the word *vulnerability* refers to a weakness or design flaws in the system. CERT is a center of internet security expertise, located at the Software Engineering Institute, a federally funded research and development center operated by Carnegie Mellon University. One of CERT's roles is to keep track of the number of vulnerabilities reported each year.

B. Problem

1. Compile and run the above *annoying* program.

```
#include <iostream>
using namespace std;

int main()
{
     while (1)
         cout << '\a' ;
}
```

2. Check out CERT's vulnerability stats from **http://www.cert.org/stats/cert_stats.html#vulnerabilities**
   1. Create a file called "cert1.txt" that includes the year and the number of vulnerabilities. Use spaces to separate your data. You can use the Visual C++ editor, Word or Notepad to create the file. When you save the program choose the "text file" option and name your file "cert1" because Word or Notepad will automatically append the extension ".txt". Put this file in the *C:\temp* folder or in some place that is easy to find.
   2. Write a C++ program that reads the file "cert1.txt" prints the year and vulnerabilities to the screen and calculates the total vulnerabilities.
   3. Modify the above program to also calculate the % increase or decrease for each year. Write year, # of vulnerabilities, and %

increase/decrease (all formatted) to a file called "cert2.txt"
    4. Summarize the trend and give your explanation for increases or decreases.
3. One way of securing passwords is to prohibit users from entering passwords that are too similar to previous passwords.
    1. Write a program that asks the user to create a password. The password cannot be the same as the previous five passwords (stored in a file created by you). When they enter a valid password, store this into another file along with the answer to two security questions. Turn in your program and files along with the answers to the following questions:
    2. List different ways to secure passwords.
    3. Can you think of some problems with these password restrictions?
4. A PIN or Personal Identification Number is typically comprised of 4 numeric digits. Write a program called GuessPIN that keeps looping until a valid 4-digit PIN is entered. Use a while loop. (This is similar to the Password program you wrote before.)
    1. Modify the above program to allow only 3 guesses.
    2. How many four-digit PINs are possible?
5. Write a program to generate all possible PINs (use nested for loops) and store into a file.
6. Write a program called CrackPIN that guesses a PIN by using all the PINs from the file.

C. Checklists

Complete the checklist in *Figure 6* for each program above.

D. Analysis

1. Why would someone write a virus or worm?
2. Write a paragraph summarizing your findings about the security of PIN numbers.

Turn in programs (marked after completing checklist), output and questions.

4.10 CS1 Function Lab

A. Background
*Primitives***:** Functions
Security: hacker, black-hat, white-hat

1. Describe the following terms:
   a)Hacker b) Black-hat security c)White-hat security

B. Problem
instructor supplied

C. Checklist

Complete the preceding checklist for each program.

Security Checklist
1. For each integer:
• Mark with V any assignment operation (check all =) that could potentially overflow (multiplication is particularly risky)
• Mark with V any input (check all cin >>)
• Mark with a V division that is not validated (divide by zero could occur)
2. For all input:
• Mark with V any input that is not immediately validated (check all cin >>)
• Mark with V any function arguments that are not validated.

Figure 7: Security Checklist – CS1.2

4.11 CS1 Array Lab

A. Background
Primitive: arrays
Security: Buffer Overflow

A **buffer overflow** occurs when data is written beyond the boundaries of a fixed length buffer overwriting adjacent memory locations which may include other buffers, variables and program flow data.
Risk: Writing outside the bounds of a block of allocated memory can corrupt data, crash the program, or cause the execution of malicious code.WARNING: over 80% of security problems result from buffer overflows!
Real world Example: The earliest known exploitation of a buffer overflow was the Morris worm in 1988. In 2001, the Code Red worm exploited a buffer overflow in Microsoft's Internet Information Services and in 2003 the SQLSlammer worm compromised machines running Microsoft SQL Server 2000.

B. Problem

```
// buffer overflow demo
#include <iostream>
using namespace std;
int main()
{
    int importantData = 1;
    int buffer[10];

    cout << "importantData = " << importantData << endl;
    cout << "buffer overflow " << endl;
    for (int i = 0; i < ??; i++)
        buffer[i] = 7;
    cout << "importantData = " << importantData << endl;
    return 0;
}
```

1. Type in the above program. What value should replace the ?? Compile and Run with this value.
2. Replace the ?? with 20 this time. What happens? Why?

C. Checklist

Security Checklist		
1. For each integer:		
● Mark with V any assignment operation (check all =) that could potentially overflow (multiplication is particularly risky)		
● Mark with V any input (check all cin >>)		
● Mark with a V division that is not validated (divide by zero could occur)		
2. For all input:		
● Mark with V any input that is not immediately validated (check all cin >>)		
● Mark with V any function arguments that are not validated.		
3. For each array reference:		
● Mark with V any assignment that could result in an overflow. EXA: arr[i] = 0; // could i >= ARR_SIZE?		
● Mark with V any loop that has a variable boundary EXA: for (i = 0; i <= riskyInt; i++) array[i] = 0;// is riskyInt < ARR_SIZE?		
● Mark with V any unvalidated indices that are input EXA: cin >> i; array[i] = x; // is i < ARR_SIZE?		
● Mark with V any dangerous use of string functions within range? (hint: do not use: gets, strcpy, sprintf)		
For each V:	Y	N
● Is the value used as an array index? EXA: array[riskyInt] = x;		
● Is the value used as the bound of an array? EXA: for (i=0;i<riskyInt;i++) array[i] = x;		
● Is the value used in security critical code?		
Shaded areas indicate High Risk!		
Mitigate	High Risk areas require validation Strong typing – use unsigned if possible. Only accept valid data and reject everything else	

Figure 8: Security Checklist – CS1.3

D. Analysis

1. Describe the buffer overflow problem.
2. Give three real life examples of buffer overflow attacks (research on the web).
3. What can result from a buffer overflow?
4. List three ways you could potentially overflow a buffer in your program.
5. How could you prevent a buffer overflow from occurring in your program?
6. Please provide feedback on the checklist.
7. Please submit the marked up program and your answers.

4.12 CS2 Class Lab1

A. Background

Primitive: Classes
Security: Vulnerabilities, integer overflow, buffer overflow, input validation
In computer security, the word **vulnerability** refers to a weakness in a system allowing an attacker to violate the integrity, confidentiality, access control, availability, consistency or audit mechanisms of the system or the data and applications it hosts.
In early 2007, the SANS Institute analyzed all critical security vulnerabilities discovered and reported during 2006 seeking to identify the specific programming errors that were the root causes of those vulnerabilities. The research team found three programming errors that were responsible for more than 85% of those critical vulnerabilities. These three errors were:

1. Input validation
2. Integer overflow
3. Buffer overflow

Describe each of the above errors and explain how they could occur in your programs.

B. PROBLEM:

Instructor supplied.

C. Checklist

Complete the checklist in *Figure 8* for each program above.

D. Analysis

1. Explain the following statement: Most security violations are the results of software vulnerabilities.
2. Why is security a major concern in the computer industry today?
3. Explain the difference between security software and software security.

4.13 CS2 Dynamic Data

A. Background

Read the article at http://www.securityfocus.com/news/11477 and briefly summarize.

Security Mantra: All Input is Evil!

- most (if not all) active attacks are the result of some type of input from an attacker
- Secure programming - ensure that inputs from bad people do not do bad things
- Defensive technique -> Input validation

B. Problem

Design a class that has an array of floating point numbers. The constructor should accept an integer argument and dynamically allocate the array to hold that many numbers. The destructor should free the memory held by the array. In addition, there should be member functions to perform the following operations:

- Display the entire array
- Store a number in any element of the array (validate the index)
- Retrieve an number from any element of the array (validate)
- Return the highest value stored in the array
- Return the lowest value stored in the array
- Return the average of all the numbers stored in the array (what validation is necessary here?)

Write client code to demonstrate each of the above functions. Create a second instance of an array and use the assignment operation and pass it to a function to indicate why a copy constructor is necessary. Add a copy constructor and a member function for = and demonstrate that they work correctly.

C. Checklist

For the program above, complete the following checklist.

Security Checklist		
1. For each integer:		
● Mark with V any assignment operation (check all =) that could potentially overflow (multiplication is particularly risky)		
● Mark with V any input (check all cin >>)		
● Mark with a V division that is not validated (divide by zero could occur)		
2. For all input:		
● Mark with V any input that is not immediately validated (check all cin >>)		
● Mark with V any function arguments that are not validated.		
3. For each array reference:		
● Mark with V any assignment that could result in an overflow. EXA: arr[i] = 0; // could i >= ARR_SIZE?		
● Mark with V any loop that has a variable boundary EXA: for (i = 0; i <= riskyInt; i++) array[i] = 0;// is riskyInt < ARR_SIZE?		
● Mark with V any unvalidated indices that are input EXA: cin >> i; array[i] = x; // is i < ARR_SIZE?		
● Mark with V any dangerous use of string functions within range? (hint: do not use: gets, strcpy, sprintf)		
For each V:	Y	N
● Is the value used as an array index? ● EXA: array[riskyInt] = x;		
● Is the value used in pointer arithmetic? ● EXA: ptr1 = ptr1 + riskyInt;		
● Is the value used as the bound of an array? ● EXA: for (i=0;i<riskyInt;i++) array[i] = x;		
● Is the value used as the size of a dynamic object? EXA: stuff = new(riskyInt);		
● Is the value used in security critical code?		
● Are arrays close to pointers? (Hint: remember the return address of a function is a pointer)		
Shaded areas indicate High Risk!		
Mitigate	High Risk areas require validation Strong typing – use unsigned if possible. Only accept valid data and reject everything else	

Figure 9: Security Checklist – CS2.1

4.14 CS2 Inheritance Lab

A. Background

Vulnerability-A weakness in a computing system that can result in harm to the system or its operations, especially when this weakness is exploited by a hostile person or organization or when it is present in conjunction with particular events or circumstances

Read the following article:http://www.sans-ssi.org/top_three.pdf

1. Describe the three programming errors most frequently responsible for critical security vulnerabilities and security incidents in 2006.

B. Problem

Instructor supplied

C. Checklist

Complete the checklist in *Figure 9* for each program above.

D. Analysis

1. How could you fix (mitigate) the vulnerabilities above?
2. Which fix is easy and which is not?
3. Discuss the risk in not fixing each of the vulnerabilities.

4.15 SYSANAL Risk Analysis Lab

A. Background

Key to software security is risk management. The risk management process is a full lifecycle activity that includes identifying, synthesizing, ranking, and keeping track of risks throughout software development for both large and small projects.

B. Problem

1. Complete a security risk analysis for your company.
2. Identify three of your company's **Assets,** (could be tangible or intangible)
   a. Examples: Customer Database, Organization reputation, Building, Web server, intellectual property, Funds
3. For each asset, identify a possible **Threat** (negative thing that could happen, could be intentional, accidental, natural)
   a. Examples: flood, theft, hacker, service unavailability
4. List **Vulnerabilities**: weaknesses that make a system more prone to attack or an attack more likely to succeed
5. Determine:
   a. **Probability** – likelihood of occurrence, 1: Low, 2: Med, 3: High
   b. **Harm** – damage inflicted, 1: Low, 2: Med, 3: High
   c. **Risk** = Probability x Harm, 1-3: Low, 4-6: Med, 7-9: High
6. Describe steps to mitigate or control the risk

Example:

Asset	**Threat**	**Vulnerability**	**Prob**	**Harm**	**Risk**	**Mitigate**
Data Center	flood	Proximity to river	0	3	NIL	Not in 100 year plan
System administrator	absence	Lack of cross training	2	2	HIGH	No funding, risk accepted
Web server	Disk crash	Insufficient backup	1	3	MED	Daily data backup. Spare hardware on site
Customer data	Theft	Communication channel security	1	3	MED	Backup, Firewall, encryption, Data protection Standard requires encryption for external communication
Organization reputation	Server unavailability	External internet interfaces	3	2	HIGH	Firewall, Dual servers

Figure 10: Risk Analysis Sample 1

4.16 DBASE Risk Analysis Lab

Complete a security risk analysis for your database.

1. For each asset, identify three possible **Threats** (negative thing that could happen, could be intentional, accidental, natural)
    a. Examples: flood, theft, hacker, service unavailability
7. List **Vulnerabilities**: weaknesses that make a system more prone to attack or an attack more likely to succeed
8. Determine:
    a. **Probability** – likelihood of occurrence, 1: Low, 2: Med, 3: High
    b. **Harm** – damage inflicted, 1: Low, 2: Med, 3: High
    c. **Risk** = Probability x Harm, 1-3: Low, 4-6: Med, 7-9: High
9. Describe steps to mitigate or control the risk

Example:

Asset	**Threat**	**Vulnerability**	**Prob**	**Harm**	**Risk**	**Mitigate**
Customer data	flood	Proximity to river	0	3	NIL	Not in 100 year plan
	absence	Lack of cross training	2	2	HIGH	No funding, risk accepted
	Disk crash	Insufficient backup	1	3	MED	Daily data backup. Spare hardware on site
	Theft	Communication channel security	1	3	MED	Backup, Firewall, encryption, Data protection Standard requires encryption for external communication
	Server unavailability	External internet interfaces	3	2	HIGH	Firewall, Dual servers

Figure 11: Risk Analysis Sample 2

4.17 DBASE SQL Injection

A. Background

SQL injection is a type of security exploit in which the attacker adds SQL code to a Web form input box to gain access to resources or make changes to data. Many Web forms have no mechanisms in place to block input other than names and passwords. Unless such precautions are taken, an attacker can use the input boxes to send their own request to the database, which could allow them to download the entire database or interact with it in other illicit ways. The risk of SQL injection exploits is on the rise because of automated tools.

Example 1: **SELECT id FROM logins WHERE user = '$user' AND pswd = '$pswd'**

If the variables $user and $pswd are requested directly from the user's input, this can easily be compromised. Suppose that we gave "Joe" as a username and that the following string was provided as a password: anything' OR 'x'='x

SELECT id FROM logins WHERE user = 'Joe' AND pswd = 'anything' OR 'x'='x'

As the inputs of the web application are not properly sanitized, the use of the single quotes has turned the WHERE SQL command into a two-component clause. The 'x'='x' part guarantees to be true regardless of what the first part contains. This will allow the attacker to bypass the login form without actually knowing a valid username / password combination!

Example 2: **SELECT * FROM OrdersTable WHERE ShipCity = '" + ShipCity + "'"**

The user is prompted to enter the name of a city. Assume that the user enters the following: Redmond'; drop table OrdersTable

In this case, the following query is assembled by the script:

SELECT * FROM OrdersTable WHERE ShipCity = 'Redmond';drop table OrdersTable--'

The semicolon (;) denotes the end of one query and the start of another. When SQL Server processes this statement, SQL Server will first select all records in OrdersTable where ShipCity is Redmond. Then, SQL Server will drop OrdersTable.

Real Life Examples:

Jan. 2008: Mass SQL injection attack compromises 70,000 websites

2006: One of the more prominent alleged attacks using SQL injection occurred earlier this year when the University of Southern California's student application system was hacked and compromised more than 270,000 records after a hacker exploited a flaw in the admissions department's SQL database to bypass authentication.

Prevention: Use Database stored procedures, parametrized queries, Validate data: All input is evil. Be strict about valid input and reject everything else., Principle of least-privilege. limit the permissions granted to the database user account the Web application is using.

4.18 Programming Languages Type Safety

A. Background:

Type checking-The process of verifying and enforcing the constraints of types, may occur either at compile-time (a static check) or run-time (a dynamic check).

Type strength – strong/weak: strongly typed involves not allowing an operation to succeed on arguments which have the wrong type. Weak typing means that a language implicitly converts (or casts) types when used.

Type safety-Safely and unsafely typed systems. Computer scientists consider a language "type-safe" if it does not allow operations or conversions which lead to erroneous conditions.

B. Problem

Chose 5 programming languages and fill out the following chart:

Language	Type checking	Type strength	Type safety

Figure 12: Programming Language Type Safety

4.19 Programming Languages Vulnerabilities

A. Background:

In computer security, the word **vulnerability** refers to a weakness in a system allowing an attacker to violate the integrity, confidentiality, access control, availability, consistency or audit mechanisms of the system or the data and applications it hosts.

In early 2007, the SANS Institute analyzed all critical security vulnerabilities discovered and reported during 2006 seeking to identify the specific programming errors that were the root causes of those vulnerabilities. The research team found three programming errors that were responsible for more than 85% of those critical vulnerabilities. These three errors were:

1. Input validation
2. Integer overflow
3. Buffer overflow

Describe each of the above errors and explain how they could occur in your programs.

B. Problem

instructor supplied

C. Checklist

Use the checklist found in Figure 9 to check your program for vulnerabilities.

5. Results

Security integration is ongoing in CS0, CS1, and CS2 at Towson University. Additionally, security integration has begun in Systems Analysis and Design, Programming Languages: Design and Implementation, and Database Management (COSC and CIS). The following chapter details the results of the study conducted spring 2007 and fall 2007.

5.1 Assessment

In spring 2007, security integration was implemented in all sections of CS0 and two of four sections of CS1. For the purpose of analysis, all courses identified as security integrated are identified with an 'S': CS0S, CS1S, and CS2S. Non-security integrated sections (indicated by CS0, CS1, or CS2) are considered controls in this study. 150 students took the pretest and 109 students took the posttest as detailed below:

		pretest	**posttest**
CS0S	COSC175.001	19	20
CS0S	COSC175.002	14	6
CS0S	COSC175.101	17	10
CS1S	COSC236.001	19	16
CS1	COSC236.002	27	14
CS1	COSC236.003	19	17
CS1S	COSC236.004	27	13
CS1	COSC236.101	8	13
TOTAL		150	109

Table 4: *Number of students by sections – spring2007*

In fall 2007, security integration was repeated in one of three sections of CS0, three of four sections of CS1, and one of four sections of CS2. 242 students took the pretest and 166 students took the posttest as detailed below:

		pretest	**posttest**
CS0A	COSC175.001-2	43	41
CS0S	COSC175.101	21	18
CS1A	COSC236.001	20	10
CS1SB	COSC236.002	19	8
CS1SA	COSC236.003	28	18
CS1SC	COSC236.004	25	13
CS1B	COSC236.101	18	12
CS2A	COSC237.001	23	9
CS2B	COSC237.002	12	14
CS2C	COSC237.003	18	12
CS2S	COSC237.101	15	11
Total		242	166

Table 5: *Number of students by sections– fall 2007*

A total of 392 students took the security pretest and 275 students took the posttest.

	spring 2007		**fall 2007**		**total study**	
	pretest	posttest	pretest	posttest	pretest	posttest
cs0			43	41	43	41
cs0s	50	36	21	18	71	54
cs1	54	44	38	22	92	66
cs1s	46	29	72	38	118	67
cs2			53	35	53	35
cs2s			15	12	15	12
TOTAL	**150**	**109**	**242**	**166**	**392**	**275**

Table 6: *Number of students: security integrated/control*

5.2 Demographics

The population we tested was approximately 27% female and 23% non-white.

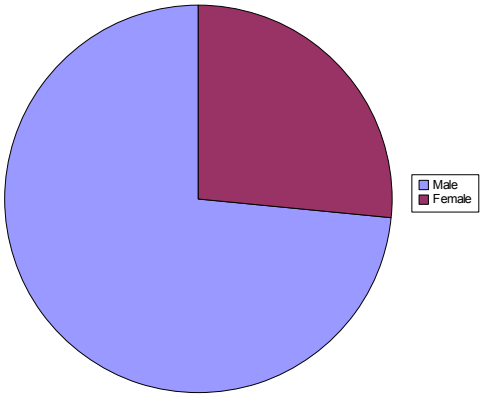

Figure 13: Gender

	Pre		Post	
	#	%	#	%
male	287	73.2	202	76.5
female	105	26.8	73	23.5
total	392	100	275	100

Table 7: *Gender*

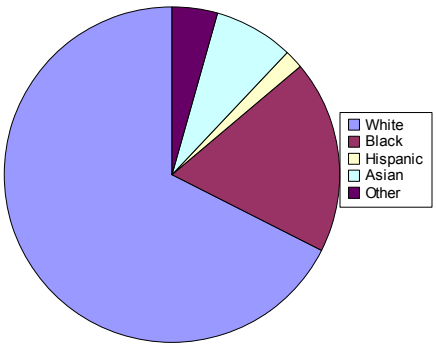

Figure 14: Ethnicity

	Pre		Post	
	#	%	#	%
White	249	64.2	185	67.5
Black	71	18.3	51	18.6
Hispanic	6	1.5	5	1.8
Asian	42	10.8	21	7.7
Other	20	5.2	12	4.4
total	392	100	275	100

Table 8: *Ethnicity*

Only 8% of the participants were seniors, 36% were freshmen, 32% were sophomores, and 23% were juniors. 68% of the students were 20 years or younger, 26% were between 21 and 25, 2% were 26-30 and 3% was over 30.

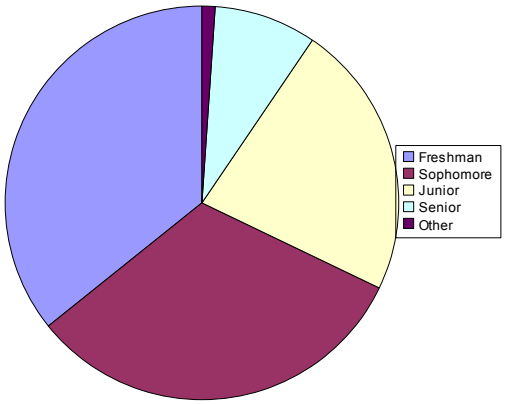

Figure 15: Class

	Pre		Post	
	#	%	#	%
Freshman	115	29.3	98	35.6
Sophomore	103	26.3	88	32
Junior	97	24.7	62	22.5
Senior	30	7.7	23	8.4
Other	9	2.3	3	1.1
Total	354	90.3	274	99.6

Table 9: *Class*

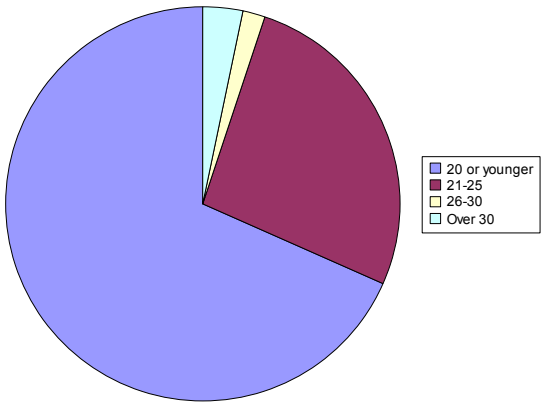

Figure 16: Age

	Pre		Post	
	#	%	#	%
20 or younger	264	67.3	188	68.4
21-25	105	26.8	73	26.5
26-30	10	2.6	5	1.8
Over 30	13	3.3	9	3.3
Total	392	100	275	100

Table 10: *Age*

Most of the students were COSC majors (35%) and CIS majors (34%), over 30% were undecided (3%) or other majors.

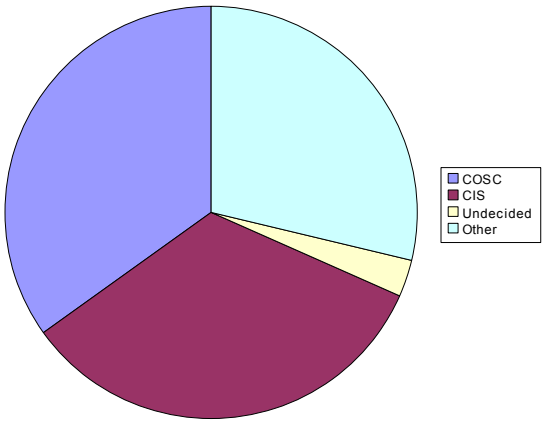

Figure 17: Major

	Pre		Post	
	#	%	#	%
COSC	147	37.6	96	34.9
CIS	131	33.5	92	33.5
Undecided	18	4.6	8	2.9
Other	95	24.3	79	28.7
Total	391	100	275	100

Table 11: *Major*

5.3 Reliability Analysis

The test used for assessing security knowledge was measured to determine its reliability as an instrument. The security test was determined to have a Cronbach alpha score of .74. Cronbach's alpha is a measure of how well each individual item in a scale correlates with the sum of the remaining items. It measures consistency among individual items in a scale. A value of .74 is considered satisfactory.

The security test was created using a multiple-choice format, see Appendix B, and included 5 general security questions and 15 questions specifically related to the principles integrated in the curriculum. The mean score for all 667 tests was .67 with a standard deviation of .18.

5.4 Security Scores

The format of the study design was the Pretest-Posttest Control Group Design. In this experimental design, specific sections were designated to receive security integration and the remaining sections served as controls. Both groups were given pretests and posttests. Group A, the control group, includes sections CS0, CS1, and CS2. Group B, the security group, includes sections CS0S, CS1S, and CS2S.

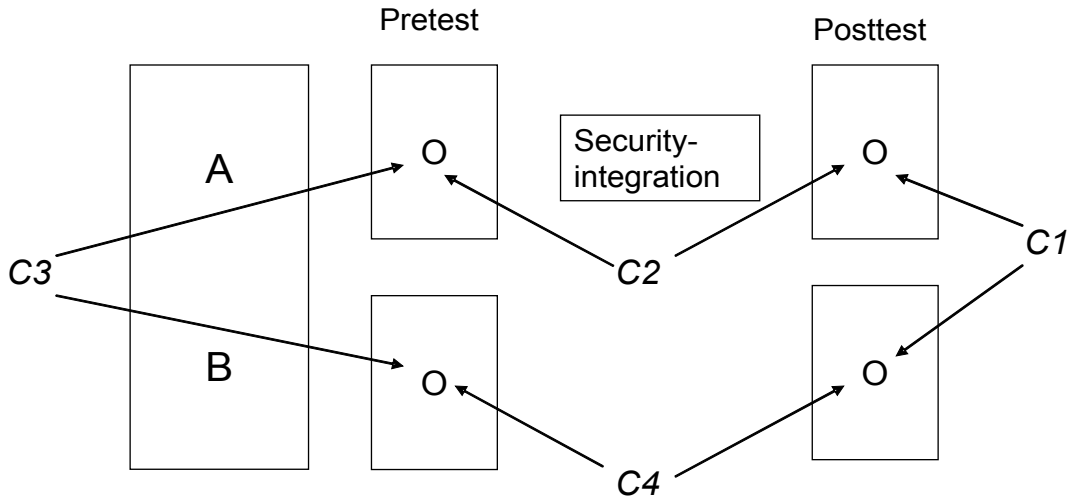

Figure 18: Pretest-Posttest Control Group Design

The following comparisons were conducted:

- Comparison 1 shows how the security group posttest differs from the control group
- Comparison 2 & 4 show how the two groups changed from pretest to posttest in terms of difference or "gain"
- Comparison 3 compares the pretest scores and indicates whether or not the groups were equivalent to begin with

- Comparison 4 also indicates whether or not there is any difference over time for the control group.

5.4.1 Comparison 1: Control posttest to Security posttest

The total posttest mean for the security integrated group (M = .74, N = 133) was higher than the control group (M = .66, N = 142). The mean difference between conditions was .08 and the 95% confidence interval for the estimated population difference is between -.12 and -.04. An independent t-test showed that the differences between conditions was very significant (t = -3.702, df = 273, p < .001).

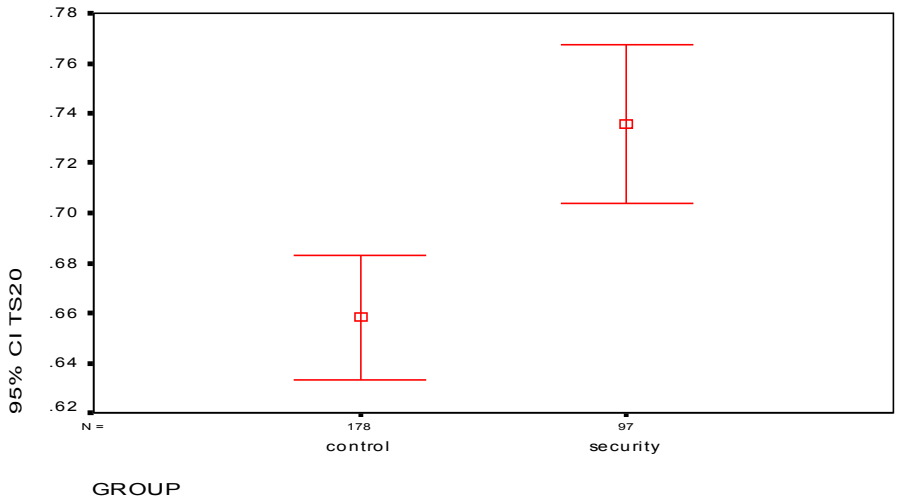

Figure 19: Error Bar: control vs security posttest score

5.4.2 Comparison 2: Security pretest to posttest

The scores for the security group significantly improved (p = .005) from the pretest (M = .68, N = 204) to the posttest (M = .74, N = 133) by .06, or 9%.

5.4.3 Comparison 4: Control pretest to posttest

The slight improvement for the control group between the pretest (M = .64, N = 188) and posttest mean (M = .66, N = 142), of .02, or 3%, was found to be statistically insignificant.

The charts below shows that overall, security scores improved for the security group.

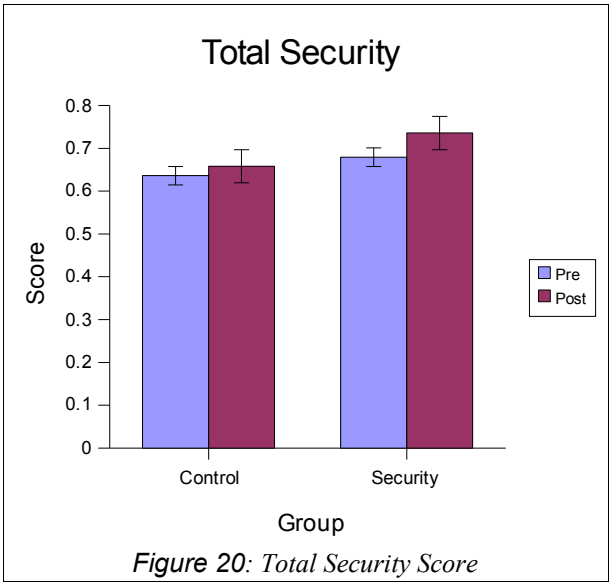

Figure 20: Total Security Score

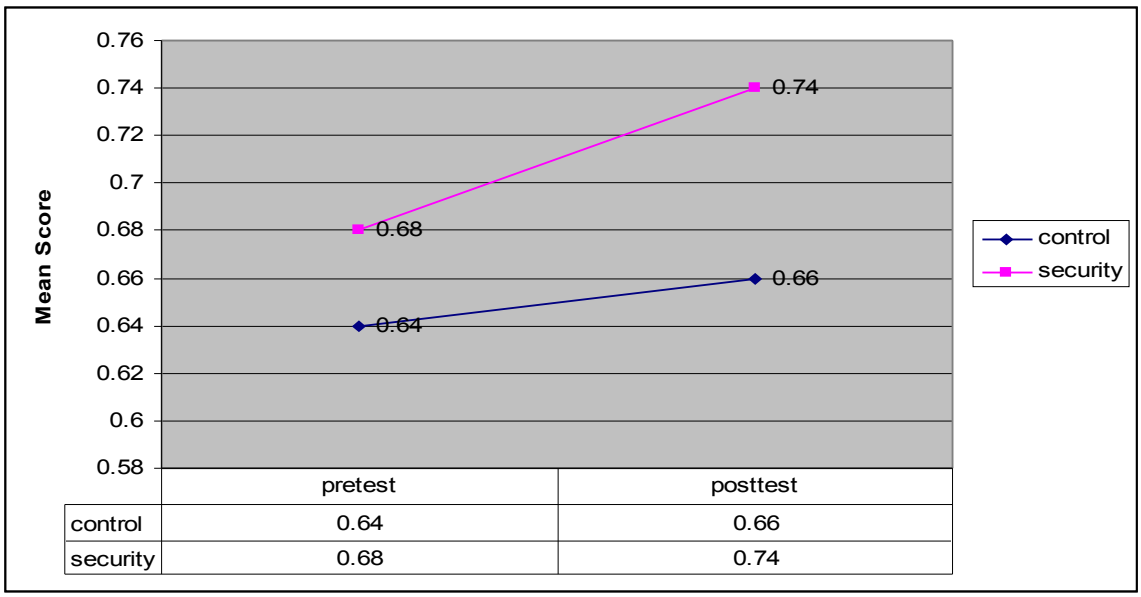

Figure 21: Gain Scores

5.4.4 Comparison 3: Control pretest to Security pretest

The score for the security group's pretest (M = .66, N = 204) was slightly higher than the control group's pretest (M= .64, N = 188), but the difference was found to be insignificant.

5.4.5 Comparison 4: Control pretest to posttest

The slight improvement for the control group between the pretest (M = .64, N = 188) and posttest mean (M = .66, N = 142), of .02, or 3%, indicates there was no significant improvement over time.

5.4.6 By Class

The scores by class, CS0, CS1, and CS2, are depicted graphically below:

Figure 22: Total Security Score by class

The table below breaks the security scores down by section. All sections showed increased scores in the posttests and scores increased more for the security groups than in the control groups.

	PRE	POST	Gain	%gain
Cs0	0.59	0.63	0.04	6.92%
Cs0s	0.49	0.54	0.05	10.38%
Cs1	0.65	0.67	0.02	2.34%
Cs1s	0.71	0.77	0.06	9.06%
Cs2	0.69	0.70	0.01	1.79%
Cs2s	0.74	0.85	0.10	13.79%

Table 12: *Gain scores*

5.4.7 By Study

The results for Study 1 (Spring 2007) are shown below. The security scores for CS01, where all sections were integrated, increased by 12% from pre to post. The security scores for the control group for CS1 did not change from pre to post, but in the security sections, the scores increased 7.65%.

	PRE	POST	Gain score	Percent
cs0s	0.62	0.69	0.07	12.02%
cs1	0.64	0.64	0.00	-0.37%
cs1s	0.72	0.78	0.06	7.65%
total	0.66	0.69	0.04	5.31%

Table 13: *Study1: Gain scores*

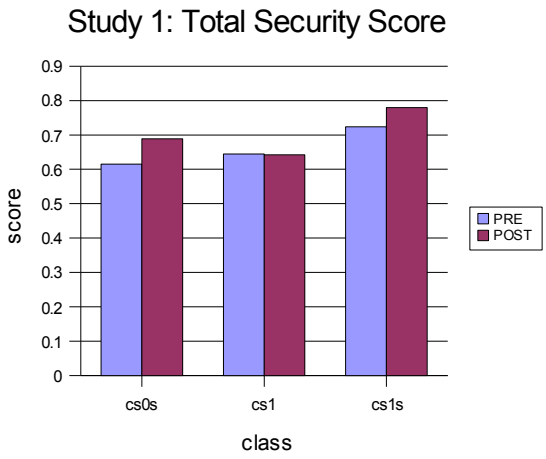

Figure 23: Study 1: Total Security Scores

The results for Study 2 (Fall 2007) are shown below. The security scores for CS0, where one section was integrated, increased by 10% from pre to post. The security scores for the control group and the security group for CS1 both increased and in CS2 the scores in the security section increased by almost 14% as compared to less than 2% in the control sections.

	PRE	POST	Gain score	Percent
cs0	0.57	0.58	0.02	3.29%
cs0s	0.49	0.54	0.05	10.38%
cs1	0.66	0.71	0.05	8.13%
cs1s	0.69	0.76	0.07	9.80%
cs2	0.69	0.70	0.01	1.79%
cs2s	0.74	0.85	0.10	13.79%
Total	0.65	0.68	0.03	4.75%

Table 14: *Study 2: Gain scores*

Figure 24: Study 2: Total Security Scores

5.5 Qualitative Results

In general, students and instructors in the security-integrated sections reacted positively to the security topics. Most answers to research and discussion questions were thorough and well thought out. To demonstrate student feedback in the analysis and discussion section of the lab modules, several questions and a sampling of answers are listed below:

Question: Explain the trend in vulnerabilities as reported by CERT.
Increases might be because more features have been added to computer systems and have not been fully checked because companies like Microsoft are anxious to get them on the market by deadline. *The trend is that as technology with computers grow, more hackers, viruses and intrusions can occur....decreases with greater security measures* *I believe the reasons for this increase are an increase in the knowledge of hackers. The more they learn and know, the more they will be able to get into your compute.*

Table 15: *Sample Question & Answers 1*

Question: Can you think of some problems with password restrictions?
*Difficult to remember intricate passwords ****** *Security questions may be known by others* *Hard not to repeat passwords* *The problem with really secure passwords is that they can be quite complicated and hard to remember.*

Table 16: *Sample Question & Answers 2*

Question: *Write a paragraph summarizing your findings about the security of PIN numbers.*
Initially I thought that four digit PINs were very secure. I have used them in banking and with credit cards and I always thought there was no way for someone to crack my PI, especially since there are 10,000 different combinations. However, after writing the program that cracked a PIN I now know that PINs are not very secure. If someone like myself with very little computer training could write a code that cracked a PIN then it is scary to think what a very savvy programmer could do. *This lab has shown me that PIN numbers are not very secure. They are easily cracked because there are not that many different options it could be. Pins could be made more secure by having a limit on the number of guesses. Also, if a pin number had more digits than 4, then it would be more secure because there are more possibilities… This is upsetting about the lack of security of PIN numbers because someone could easily crack the pin number for my ATM card.*

Table 17: *Sample Question & Answers 3*

Instructors in the security-related sections were eager to incorporate security in their courses and several introduced security material beyond the modules that were provided to them.

The risk analysis exercise (Figure 10) was introduced this semester in the Systems Analysis and Design and the instructor described this as the "best class exercise" she has ever administered. The students worked in teams; the results are below:

Asset	**Threat**	**Vulnerability**	**Prob**	**Harm**	**Risk**	**Preventive**	**Mitigate**
Inventory Database	Hackers	Outdated security	1	3	Low	Update security	Daily backups
Building	Natural disasters or fire	Poor maintenance	2	3	Med	Sprinkler System	Insurance
Products	Theft	Inadequate alarm system	3	3	High	New alarm with motion & inventory sensors	insurance

Table 18: *Risk Analysis Exercise, Team 1*

Asset	**Threat**	**Vulnerability**	**Prob**	**Harm**	**Risk**	**Preventive**	**Mitigate**
Customer database	System crash	No backup	1	2	low	Back-up server	backup
Building	fire	oven	2	3	Med	High quality ovens	Clean daily
Food (rest-aurant)	Spoiled or poi-soned	Improper care	2	3	med	Secure food, don't receive food from outside sources	Keep food are clean

Table 19: *Risk Analysis Exercise, Team 2*

Asset	Threat	Vulnerability	Prob	Harm	Risk	Preventive	Mitigate
contact	Too busy, unwil-ling to help	Delay project, lack of information	2	2	med	Alternate contact	Multiple ways to get in contact
Intel-lectual property	POS com-pany not allow-ing plug-in	Delay project	1	2	low	Contact POS manufac-turer	Design stand alone system
Building power	Power outage	System wouldn't work	1	2	low	Back up generator	Off site generator, hard copy backup

Table 20: *Risk Analysis Exercise, Team 3*

Asset	Threat	Vulnerability	Prob	Harm	Risk	Preventive	Mitigate
Customer db	crash	Backup security	2	3	med	Mainte nance quality freq. Sw updates	Daily backup firewall
inventory	Theft natural disaster	Insufficient security	2	3	med	Good security (guards)	Insure inventory
building	natural disaster	Uninsured property	1	3	low	Prepare building for natural disasters	Insure inventory

Table 21: *Risk Analysis Exercise, Team 4*

Asset	Threat	Vulnerability	Prob	Harm	Risk	Preventive	Mitigate
Ordering system	BOTS	Ridiculous ordering	low	high	low	CAPTCH A	Reasonable check
security	hackers	Not good security	med	high	med	High quality security system	Insurance coverage

Table 22: *Risk Analysis Exercise, Team 5*

Asset	Threat	Vulnerability	Prob	Harm	Risk	Preventive	Mitigate
Organizat ion repu-tation	accident	Public image damaged, loss of customers	2	2	med	Safety assessment customer service planning	Publicize recovery
Building & sites	In-creased rent or lease, damage (fire) re-location	No disaster recovery	1	2	low	backup info offsite	Ability to relocate insurance
funds	Overspe nding, running out of project funds	Bankruptcy overspending	3	3	high	Accurate budgeting	Financial analysts capitol to bank errors

Table 23: *Risk Analysis Exercise, Team 6*

6. Discussion

6.1 Summary

This purpose of this project was to develop, implement, and evaluate an integrated security curriculum model for students in the Computer and Information Sciences department at Towson University. Security modules were developed and subsequently delivered during spring 2007 and fall 2007 across selected sections of the core courses: CS0, CS1, and CS2. The modules were laboratory-based to allow seamless adoption and integration and included the innovative use of checklists to promote active learning and encourage critical thinking. The security group increased their knowledge by .06 points or 9%. This significant ($p = .005$) improvement in security knowledge indicates that security lab modules are an effective way to teach secure coding and design principles to more students, earlier in their studies, with minimal impact on existing curricula.

A two group control group experimental design was employed using pretests and posttests for evaluation. Ten sections of CS0, CS1, and CS2 were security-integrated and the remaining ten sections were controls. A security assessment was administered to students in all sections of CS0, CS1, and CS2 at the beginning and end of each semester. A total of 392 students completed pretests and 275 students completed posttests. The reduced number of posttest respondents is attributed to normal attrition and declining attendance at the end of the semester.

Prior experience with these courses indicates that the demographics of the sample is representative. The demographics of the pre and post-groups were almost

identical. Most students were white (77%) and male (27%). The second largest ethnic group represented was African-Americans at 18%. Most students were 20 years or under (68%) and freshmen (36%), which is expected as these courses are generally taken during the first 2 years of study. Students were fairly evenly distributed across COSC, CIS, and other. CS1 is required for Math and Physics majors.

The reliability of the security assessment instrument, which included 20 multiple-choice questions, was determined to be Cronbach alpha = .74. Attempts to analyze subsections of the test, for example, the five general security questions or the 15 targeted questions, was found less reliable, so analysis was limited to one security score. The mean score for all tests was .67 or 67% (N = 667, std = .18) which correlates to 13.4/20 correct answers.

The design of this study, a pretest-posttest control group design, was chosen because the use of pretest scores helps to reduce error variance, or the effectiveness of the test at measuring differences. There are a few disadvantages to administering pretests. External events, such as class or work experiences outside the specific courses, could affect posttest scores. Students could use pretest questions to affect their learning(Dimitrov & Rumrill, 2003). Finally, gain scores could be affected by the practice effect or taking the test twice.

The security-integrated posttest score (M = .74, N = 133) was higher than the control group (M = .66, N = 142). While this difference was very significant ($t = -3.70$, $df = 273$, $p < .001$), because the security group had higher pretest scores than the control group, it is also necessary to look at the gain scores. In fact, the security group

significantly improved ($p = .005$) by .06 points or 9% while the control group did not significantly improve.

Examination of the data by course shows the largest gain in CS2 (13.8%) and the least gain (6.9%) for CS0. The CS2 security-integrated section was the smallest, since integration occurred during the second semester only. This was also taught by the author. To minimize the discrepancy between courses and sections and to maximize the effectiveness of the security labs, a repository of lab modules for should be created for future work.

The lab was found to be a good place for integration; it is a fluid part of the curriculum and security-related labs are an interesting and useful way to reinforce key concepts. Additionally, in examining the structure of traditional CS labs, there is a wide gap between CS labs and traditional science labs. The security modules fill this gap and foster higher-level critical thinking. The feedback from the discussion questions was found to be particularly useful to instructors. Many of the students offered insightful answers to both the background and analytical questions.

Initially, students found the checklists hard to use, but this seemed to improve with practice. An incremental approach for introducing the checklists was particularly effective for CS0 students. CS2 students seemed to grasp the concept more quickly. The use of a standard lab format was also found to be beneficial to students and instructors. Students became familiar with the different components and the process of completing the checklist became routine, a practice they can carry with them as programmers.

The risk analysis exercise, administered in the Systems Analysis and Design

and Database course, was an extremely popular exercise. Most groups quickly grasped the concept and there was a great deal of lively debate. The goal of this exercise was to introduce the rudimentary idea of risk assessment and mitigation. This concept has potential for further development and integration into other courses.

6.2 Future work

Security integration is ongoing in CS0, CS1, CS2, Systems Analysis and Design, Database Management, and Principles of Programming Languages. We have applied for funding from NSF to continue this work and expand to other two and four year institutions. There are many aspects that were brought to light through these experiments that need work and further exploration. This section highlights some of these components.

Development of lab modules, particularly the checklists, proved to be extremely time consuming. Preliminary modules and checklists were modified based on student and instructor feedback and the third iteration seemed to be the optimal format. Additionally, distribution of materials was inconvenient and could be streamlined if an online-repository was created.

Conducting the pretests and posttests online expedited data collection, but required careful planning as far as scheduling laboratory time. To ensure optimal attendance, this may be moved forward or backward in the semester.

While we were effective at increasing security knowledge for the semester, it would be interesting and relevant to assess the long term retention of the security skills by administering the assessment test in later classes and evaluating and analyzing security

scores. Additionally, in this study, our assessment was limited to testing students' knowledge and awareness. Future work should include assessment of students' ability to apply security skills. This can be done by evaluating lab work or by modifying the assessment test to include questions that include coding. The checklist, which for this study was only used for self-evaluation, can be extended for use as a scorecard by an independent grader.

6.3 Conclusions

Overall, we were successful in developing and implementing an integrated core security curriculum for undergraduate computing students. Statistically significant changes were demonstrated in students' security knowledge.

Security education has been identified as important to producing more secure software, yet, educational programs address security exclusively through tracks and security courses. Consequently, most students lack important security knowledge at graduation. To address this shortcoming, this study developed, implemented, and evaluated an integrated core curriculum model for Towson University COSC and CIS undergraduates. Our plan utilized an active learning approach in the laboratory as the main vehicle for a seamless integration. Over a 12-month period, 342 students in sections of CS0, CS1, and CS2 participated in this study. Teaching strategies included security lab modules, which extended the traditional programming assignment to move student beyond problem solving to the critical thinking skills of analysis, synthesis, and evaluation. Evaluation methods included pre and post security assessment The number of posttest responses was reduced due to attrition and declining attendance. A significant improvement in students' security knowledge was demonstrated and overall instructor

feedback was positive. Security integration in CS0, CS1, CS2, Systems Analysis and Design, Principles of Programming Languages, and Database Management is ongoing.

Appendices

Appendix A – Institutional Review Board Documents

EXEMPTION NUMBER: 07-1X67

To: Blair Taylor

From: Institutional Review Board for the Protection of Human Subjects, Deborah Gartland, Member

Date: Tuesday, May 01, 2007

RE: Application for Approval of Research Involving the Use of Human Participants

Thank you for submitting an application for approval of the research titled,

Threading Secure Coding into the Undergraduate Computer Science and Computer Information Systems Curriculum

to the Institutional Review Board for the Protection of Human Participants (IRB) at Towson University.

Your research is exempt from general Human Participants requirements according to 45 CFR 46.101(b)(2). No further review of this project is required from year to year provided it does not deviate from the submitted research design.

If you substantially change your research project or your survey instrument, please notify the Board immediately.

We wish you every success in your research project.

CC: Shiva Azadegan
✓File

Appendix B – Security Test

Security pretest

This anonymous survey is being used to collect data for a security study we are doing in our classes. The results of this survey will be kept confidential and will not affect your class standing. Please do not put your name or any other identifying marks on the survey form.

Your time and effort in completing this survey is very much appreciated. Thank you.

Section I - Demographics

This section is designed to collect general information about you. Please answer the questions by checking the appropriate box.

1. Which class and section are you in?

○ COSC175.001 (Dierbach)

○ COSC175.002 (Dierbach)

○ COSC175.101 (Bachman)

○ COSC236.001 (Zimand)

○ COSC236.002 (Hochheiser)

○ COSC236.003 (Taylor)

○ COSC236.004 (Hochheiser)

○ COSC236.101 (Zimand)

○ COSC237.001 (Zimand)

○ COSC237.002 (Kim)

○ COSC237.003 (Davani)

○ COSC236.101 (Taylor)

1A. Did you take any of the following last semester (these were designated security sections)?

○ COSC175.001 or COSC175.002 (Taylor)

○ COSC175.101 (Hong)
○ COSC236.001 (Taylor)
○ COSC236.004 (Hochheiser)
○ None of the above.

2. What is your gender?
○ Male ○Female

3. What is your age?
○ 20 years or younger
○ 21-25 years
○ 26-30 years
○31 years or older

4. What ethnic group best describes you?
○White
○ Black
○ Hispanic
○ Asian
○ Other

5. What is your current student standing?
○ Freshman
○ Sophomore
○ Junior
○ Senior
○ Other

6. What is your major?
○ Computer Science
○ Computer Information Systems
○ Undecided

○ Other

Section II - Awareness

7. What are the possible consequences of someone breaking into my computer?

○ a. I may have files deleted from my computer

○ b. I may have personal communications exposed

○ c. I may have my network connection cut off

○ d. My monitor may shatter

○ e. My computer may be used to commit a crime

○ a, b, c, and e

○ a, b, and d

○ Unsure

8. Which of the following is an example of a strong password?

○ Password

○ J*p2le04>F

○ Your real name, user name or company name

○ Unsure

9. How many guesses are required for a three digit PIN?:

○ 10*3

○ 10^3

○ 3^3

○ None of the above

○ Unsure

10. Integer overflow occurs

○ when a number exceeds the largest possible value.

○ when the run-time stack runs out of storage

○ when the bounds of an array are exceeded

○Unsure

11. Integer overflow is caused by

○ a virus

○ unchecked input or an operation such as multiplication or exponentiation

○ an array overflow

○Unsure

12. Which of the following statements is not true:

○ Passwords need to be easy to remember but hard to crack.

○ Passwords should be short and easy to remember.

○ A novice programmer could write a password or PIN cracking program.

○ Limiting the number of password or PIN attempts is a necessary security measure.

○all are true

13. Using a language such as Java ensures your code is secure.

○ True

○ False

○ Unsure

14. Phishing is:

○ a program that monitors your internet activity

○ hacking

○ fraudulent e-mail asking for personal information that can be used in identity theft.

○ unsure

15. Which of the following present a security threat to my computer?

○ Myself

○ Friends

○ Customers

○ Coworkers

○ Competition

○ Hackers

○ All of the above

○ Unsure

16. This is a set of related programs, usually located at a network gateway server, that protects the resources of a private network from other networks.

○ firewall

○ sandbox

○ rootkit

○ password cracker

○ general protection fault

○ unsure

17. This is the conversion of data into a ciphertext that cannot be easily understood by unauthorized people.

○ brute force hacking

○ tunneling

○ encryption

○ ciphertext feedback

○ cloaking

○ Unsure

18. Security Software and Software Security are the same thing.

○ True

○ False

○ Unsure

19. In developing secure systems, where does security fit in?

○ After design is complete

○ During testing

○ Before implementation

○ After implementation

○ At all phases of development

○ Unsure

20. Security vulnerabilities are the result of software bugs and flaws.

○ True

○ False

○ Unsure

21. There is no such thing as 100% secure code.

○ True

○ False

○ Unsure

22. Which programming mistake is very often found in C and C++ applications and is one of the major vulnerabilities in today's applications?

○ Undocumented code

○ Buffer overflow

○ Weak passwords

○ Compiler bugs

○ Unsure

23. Dangling pointers

○ waste memory but are difficult to exploit

○ are dangerous but occur infrequently

○ are dangerous and happen frequently

○ Unsure

24. We don't have to worry about writing secure software if we have a firewall or antivirus tools on our machine.

○ True

○ False

○ Unsure

25. Risk management includes

○ a. identifying the risks to system security

○ b. determining the probability of occurrence

○ c. determining the resulting impact

○ d. mitigating the risks

○ a and d

○ all of the above

○ Unsure

26. Our program is secure if it executes correctly for all valid input.

○ True

○ False

○ Unsure

Submit Reset

Appendix C – Sample Syllabi

COSC175 Class Schedule - Spring 2007

Wk	Date	Topic	text	HW	Lab
1	1/29	**Introduction-- Syllabus Overview/Defining the Problem**	Intro 1.1-1.3 2	HW1 - due 2/7	Learn Online Visual C++ 2005
2	2/5	**Data Representation -Variables, binary**	1.4, 9.2	HW - due 2/14	Lab1 **
3	2/12	**Algorithms/Operator Design the Solution: Pseudocode**		HW -due 2/21	Data Lab ** Ops Lab
4	2/19	**Selection Selection exercises**	3	HW -due 2/28	Sel Lab
5	2/26	**Selection cont'd Test #1**	4		Sel Lab2 **
6	3/5	**Loops Loop exercises**		HW - due 3/26	Loop Lab
7	3/12	**Loops cont'd**			Loop Lab2 **
	3/19	**Spring Break**			
8	3/26	**File Management Test 2 Review**	5		File Lab
9	4/2	**Test #2 Modularity: Functions, procedures**	7	HW - due 4/18	Mod lab1
10	4/9	**Func Exercises 1 Modularity: parameters Func exercises 2**			Mod Lab 2
11	4/16	**Arrays, structures**	6	HW - due 4/30	Integer Overflow Checklist**
12	4/23	**arrays cont'd Array Exercises**			Array Lab 1
13	4/30	**Test 3 Review/Test #3**	8		Array Lab 2 **
14	5/7	**Data Structures: Lists, trees,stacks, queues**		List exer.	Make up Labs ALL LABS due
15	5/12	**Review**			survey Grade Calc
		Final Exam: Mon. 5/ 21 10:15 AM			

** Security enhanced lab

COSC236 Class Schedule - Spring 2007

Wk	Date	Topic	Text	Lab
1	1/30	**Course Syllabus Assessment Test Introduction**	1	Learn Online Instructions Visual C++ 2005 HW- Lab1 - due 2/8
2	2/6	**Elements of C++ Programs** **Expressions, function calls, output**	2 3	Lab2 **- due 2/13
3	2/13	**Conditions – if Decision Exer.**	4	Sel Lab** due 2/27
4	2/20	**Loops More loop practice**	5	
5	2/27	**Additional control structures**	5	Loop Lab** - due 3/6
6	3/6	**File input/output** **Formatted output examples**	3	File Lab** - due 3/15
7	3/13	**Test 1 Review/Test #1**		
	3/20	**Spring Break**		
8	3/27	**Functions**	6	Function Lab 1**- due 4/4
9	4/3	**Scope and More Functions** **Intro to Recursion**	19	Function Lab 2** - due 4/11
10	4/10	**Arrays Array Exercises**	7	Array Lab 1 - due 4/18
11	4/17	**Applied Arrays**	8	Array Lab 2** - due 4/25
12	4/24	**Structures**	11	Struct Lab- BufChk - due 5/2
13	5/1	**Structures cont'd Struct exercises** **Test 2 Review/Test #2**		
14	5/8	**Classes Class Examples** **OO Exercises**	13	Class lab (handed out) 5 points survey 1 point grade calculation
15	5/15	**Review**		
		Final Exam: Tues. 5/22 8AM		

COSC237 Class Schedule - Fall 2007

Wk	Date	Topic	Text	Lab
1	8/27 M 8/29 W	**Syllabus Review**	1-10	Learn Online Lab #1
2	9/5 W	**Review cont'd**	1-10	
3	9/10	**Classes UPDATED**	11	Lab2* - due 9/19
4	9/17	**Inheritance and Composition**	12	Lab3* - due 10/1
5	9/24	**Exercises**		
6	10/1	**Practice Test / Test #1**		
7	10/8	**Pointers**	13	Pointer Exerc**UPDATED due 10/15
8	10/15	**Dynamic Data Exercises**		Lab4** - due 10/24
9	10/22	**Overloading**	14	Lab5 - due 10/31
10	10/29	**Recursion Exercises**	16	Lab-Recursion-due 11/7
11	11/5	**Linked Lists Exercises**	17	
12	11/12	**Stacks/Queues Exercises Review**	18	Lab - due 12/5
13	11/19 M 11/21 W	**Test #2** **Thanksgiving break - no class**		
14	11/26	**Algorithm Efficiency Searching & Sorting exercises**	19	Lab
15	12/3	**Trees Exercises**	20	
16	12/10	**Review Security Survey**		Grade Calculation
	Final Exam: Wed 12/12			

List of References

Apvrille, A. & Pourzandi, M. (2005). Secure Software Development by Example. *IEEE Security & Privacy*, 3 (4),10-17, July/August 2005.

Azadegan, S. & O'Leary, M. (2006). Incorporating Security Concepts into First Course. *International Workshop on Informatics Education: Bridging the University/Industry Gap.*

Azadegan, S., Lavine, M., O'Leary, M., Wijesinha, A. & Zimand, M. (2003). A dedicated undergraduate track in computer security education. *Security Education and Critical Infrastructures,* 319-331.

Azadegan, S., Lavine, M., O'Leary, M., Wijesinha, A & Zimand, M. (2003). An undergraduate track in computer security. *Proceedings of the 8th Annual Conference on Innovation and Technology in Computer Science Education*, 207-210.

Azadegan, S., Lavine, M., O'Leary, M., Wijesinha, A & Zimand, M. (2006). Undergraduate Computer Security Education: A Report on our Experiences & Learning. *Proceedings of Seventh Workshop on Education in Computer Security*.

Bishop, M. & Frincke, D. (2005). Teaching Secure Programming. *IEEE Security and Privacy,* 3(5), 54-56, Sep. 2005.

Bishop, M. & Frincke, D. (2004).Teaching Robust Programming. *IEEE Security & Privacy*, 2(2), 54-57, Mar 2004.

Bishop, M. (2006). Teaching Assurance Using Checklists. *Seventh Workshop on Education in Computer Security (WECS 7).*

Bloom, B.S. (1956). *Taxonomy of Education Objectives, Handbook 1: The Cognitive Domain.* New York: David McKay Co Inc.

Briggs, T. (2005). Techniques for active learning in CS courses. *Journal of Computing Sciences in Colleges*, 21(2), 156-165, December 2005.

CERT/CC.(2008). Vulnerability Remediation Statistics. Retrieved March 1 2008 from http://www.cert.org/stats/vulnerability_remediation.html

Conklin, W. (2006). Bottom-Up meets Top-Down: A New Paradigm for Software Engineering Instruction. *Proceedings of the 10th Colloquium for Information Systems Security Education,*131-136.

Conklin, W. & Dietrich, G. (2007). Secure Software Engineering: A new Paradigm, *Proceedings of the 40th Hawaii International Conference in System Sciences (HICCS'07).* 272.

Cress, D., Roberts, B. & Simmons, J. (2004). A Strategy to Integrate Defensive Programming into the Undergraduate Computer Science Curriculum at UMBC,Graduate Thesis.

Davis, J. & Dark, M, 2003. Teaching Students to Design Secure Systems. *IEE Security and Privacy*, 1 (2), March 2003.

Davis, N., Humphrey, W., Redwine, S., Zibulski, G., & McGraw, G. (2004). Processes for Producing Secure Software. IEEE Security & Privacy, 2 (3), May 2004, 18–25.

Dimitrov, D., & Rumrill, P. (2003). Pretest-posttest designs and the measurement of change. *A Journal of Prevention, Assessment, and Rehabilitation,* 20, 159-165.

Gerhart, Susan. (2003) Explaining the Buffer Overflow Problem: Instructional Evaluation. e-learn 2003, Phoenix, AZ.

Gilliam, D., Wolfe, T., Sherif, J & Bishop, M. (2003). Software Security Checklist for the Software Life Cycle. *Proceedings of the Twelfth IEEE International Workshops on Enabling Technologies: Infrastructures for Collaborative Enterprises.* 9-11 June 2003, 243-248.

Graff, M. & van Wyck, K. (2003). *Secure Coding: Principles and Practices*. Sebastopol, CA: O'Reilly.

Harrison, W.S., Hanebutte, N., & Alves-Foss, J. (2006).Programming Education in the Era of the Internet: A Paradigm Shift, *Proceedings of the 39th Annual Hawaii Internation Conference on System Sciences (HICSS '06)*

Hoglund, G. & McGraw, G. (2004). *Exploiting Software: How to Break Code.* Boston: Addison-Wesley

Hope, P., Lavenhar, S., & Peterson, G. (2005). Architectural Risk Analysis,retrieved on March 1, 2007 from https://buildsecurityin.us-cert.gov/daisy/bsi/articles/best-practices/architecture/10.html

Howard, M. (2004) Building More Secure Software with Improved Development Processes, *IEEE Security and Privacy*. 2(6), p 62-65.

Howard, M. & LeBlanc, D. (2003). *Writing Secure Code.* Redmund, WA: Microsoft Press.

Howard, M & Lipner, S, (2006). *The Security Development Lifecycle.* Microsoft Press.

Irvine, C.E., Chin, S. and Frincke, D. (1998). Integrating Security into the Curriculum. *IEEE Computer*, 25-30, Dec. 1998.

Keil, M., Li, L., Mathiassen, L., & Zheng, G. (2006). The Influence of Checklists and Roles on Software Practioner Rick Perception and decision Making. *Proceedings of the 39th Annual Hawaii Internation Conference on System Sciences (HICSS '06), 229.2*

Levin, I. & Lieberman, E., (2000). Developing Analytical and Synthetic Thinking in Technology Education. *Proceedings of International Conference on Technology Education*, Braunshweig, Germany.

McConnell, J. (1996). Active learning and its use in computer science. *Proceedings of the 1st conference on Integrating technology into computer science education (ITiCSE '96),* 52–54.

McGraw, G. (2006). *Software Security: Building Security In.* Upper Saddle River, NJ: Addison-Wesley.

Microsoft Coorporation. (2002). Trustworthy Computing. Retrieved on March 1, 2008 from http://www.microsoft.com/mscorp/execmail/2002/07-18twc.mspx

Meier, J.D., Mackman, A., Dunner, M., Vasireddy, S., Escamilla, R. & Murukan, A. (2003). Improving Web Application Security: Threats and Countermeasures. Retrieved on March 1, 2008 from rehttp://msdn2.microsoft.com/en-us/library/ms994921.aspx

Mullins, P., Wolfe, J., Fry, M., Wynters, E. ,Calhou, W., Montante, R. & Oblitey, W. (2002). Panel on integrating security concepts into existing computer curriculum. *Proceeding of 33rd SIGCSE technical Symposium on Computer Science Education.*

Null, L. (2004). Integrating Security across the computer science curriculum. *Journal of Computing Sciences in College.*

Perrone, L.F, Aburdene, M. and Meng, X.(2005). Approaches to undergraduate instruction in computer security. *Proceedings of the American Society for Engineering Education Annual Conference and Exhibition (*ASEE 2005).

Pierce,B. (2002). *Types and Programming Languages*. MIT Press.

Pothamsetty, V. (2005). Where Security Education is Lacking. *Proceedings of the 2nd annual conference on Information security curriculum development*, 54-58.

President's Information Technology Advisory Committee,(2005). *Cyber Security: A Crisis of Prioritization, National Coordination Office for Information Technology Research and Development.* Retrieved on March 1, 2008 from http://www.nitrd.gov/pitac/reports/20050301_cybersecurity/cybersecurity.pdf (2005

Ryan, J. & Ryan, D.(2002). Institutional and Professional Liability in Information Assurance Education. Retrieved March 1, 2008 from www.danjryan.com/Institutional %20and%20Professional%20Liability%20in%20Information%20Assurance %20Education.doc

Saltzer, J. & and Schroeder, M. (1975).The protection of information in computer systems. *Proceedings of the IEEE*, 63(9).

SANS Institute. (2007). *New Report Identifies the Three Programming Errors Most Frequently Responsible For Critical Security Vulnerabilities and Security Incidents in 2006*, retrieved March 1, 2008 from http://www.sans-ssi.org/top_three.pdf

Seacord, R. (2006). Secure Coding in C and C++ Of Strings and Integers, *IEEE Security and Privacy.*

Shumba, R., Walden, J., Ludi, S., Taylor,C., Ju, A., Frank, C., Walden, J. (2006). Integrating Secure Development Practices into a Software Engineering Course, SIGCSE 2006 Birds of a Feather: Secure Software Engineering. *Proceedings of the 37th SIGCSE Technical Symposium on Computer Science Education*, 573.

Skalka, C, (2005) Programming Languages and System Security, *IEEE Security and Privacy.*

Swiderski, F. & Snyder, W. (2004). *Threat Modeling.* Redmund, WA, Microsoft Press.

Taylor, B. & Azadegan, S. (2006). Threading Secure Coding Principles and Risk Analysis into the Undergraduate Computer Science and Information Systems Curriculum.*Proceedings of the 3rd Annual Information Security Curriculum Development Conference, 24-29.*

Taylor, B. & Azadegan, S. (2007). Using Security Checklists and Scorecards in CS Curriculum. *Proceedings of the 11th National Colloquium for Information Systems Security Education.*

Taylor, B. & Azadegan, S. (2007). Teaching Security through Active Learning. *Proceedings of Frontiers in Education : Computer Science and Engineering(FECS 2007).*

Taylor, B. & Azadegan, S. (2008). Moving Beyond Security Tracks: Integrating Security in CS0 and CS1. *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education.*

Thompson, H. & Whitaker, J (2003). *How to Break Software Security*. Addison-Wesley.

Vaughn, Jr., R. (2000). Application of security to the computing science classroom. *Proceedings of the thirty-first SIGCSE technical symposium on Computer science education*, 90-94.

Viega, J. & McGraw, G. (2002). *Building Secure Software.* Boston: Addison-Wesley,.

Walden, J. & Frank, C.(2006) Secure software engineering teaching modules. *Proceedings of the 3rd annual conference on Information security curriculum development.*

White, G. & Nordstrom, G. (1996). Security across the curriculum: using computer security to teach computer science principles. *Proceedings of the 19th Nat'l Information Systems Security Conference.*

Yasinsac, A. & McDonald, J.T. (2006). Foundations for Security Awareness Curriculum. *Proceedings of the 39th Hawaii International Conference in System Sciences.*

Curriculum Vitae

Blair Taylor

311 Chapelwood Lane

Timonium, MD 21093

EDUCATION

D.Sc. Candidate, Applied Information Technology,*Towson University*, 2008

M.S. Computer Science, *Johns Hopkins University*,1983

B.A. Mathematical Sciences, *Johns Hopkins University*,1980

APPOINTMENTS

Academic Appointments:

- **Lecturer, *Towson University*,** Jan. 1997 to present.
- **Assistant Professor, *Anne Arundel Community College***, Aug. 1989 to Jan. 1997.
- **Adjunct Lecturer, *Towson University***, Sep. 1987 to June 1990.

Industrial Employment:

- **Senior Software Engineer, *ITP Digitrol***, Owings Mills, MD, Sept. 1985 to Aug. 1989.
- **Senior Software Engineer, *REXNORD Automation***, Hunt Valley, MD, Aug 1982 to Sept. 1985.
- **Programmer Analyst, *American Totalisator***, Hunt Valley, MD, June 1979 to Aug. 1982.

PUBLICATIONS

- Taylor, B & Azadegan, S., Moving Beyond Security Tracks: Integrating Security in CS0 and CS1. *Proceedings of the 38th SIGCSE technical symposium on Computer science education,* Portland, OR, 2008.
- Taylor, B & Azadegan, S.,, Teaching Security through Active Learning, *Proceedings of Frontiers in Education: Computer Science and Engineering*, Las Vegas, NV, 2007.
- Taylor, B & Azadegan, S., Using Security Checklists and Scorecards in CS Curriculum *Proceedings of the 11th Colloquium for Information*

Systems Security Education. Boston, MA 2007.

- Taylor, B & Azadegan, S., Threading Secure Coding Principles and Risk Analysis into the Undergraduate Computer Science and Information Systems Curriculum, *INFOSECCD*, 2006.
- Dierbach, C, Taylor, B, Zhou, H. & Zimand, I., Experiences with a CS0 course targeted for CS1 success in *Proceedings of the 36th SIGCSE technical symposium on Computer science education*, March 2006.

SYNERGISTIC ACTIVITIES

1. Faculty Workshop on Secure Software Development, Orlando,FL. April 2008
2. Community Outreach Award, Towson University, CSM, 2007.
3. Presenter, MAISA, Jan. 2008.
4. Chair, Articulation Committee. Duties include:

- Participate in articulation meetings with Maryland 2 year and 4 year institutions
- Evaluate course materials for transfer
- Develop and maintain transfer programs with Maryland 2 year schools
- Develop articulation agreements with high schools
- Host Open Houses for prospective freshmen

5. NCAA advisor – advise CS and CIS student athletes
6. Student Advisor – advise over 60 CS students
7. Advisory Board, Internet Multimedia Committee, CCBC
8. Course Coordinator and Developer, General Computer Science