Proceedings of the 11th Colloquium for Information Systems Security Education
Boston University
Boston, MA June 4-7, 2007

Using Security Checklists and Scorecards in CS Curriculum

Blair Taylor and Shiva Azadegan, *Towson University*

Abstract

Industry has recognized that creating secure systems requires incorporating security concepts throughout the software development lifecycle. A similar effort is required in education, integrating security best practices and risk management into the curriculum. At Towson University, we are developing and implementing a model to thread security throughout our computer science curriculum. Key to our plan is the use of security checklists and scorecards. Checklists provide a quantifiable list of security criteria to aid in writing secure code and reinforce security principles. Additionally, scorecards and checklists provide a consistent means of evaluation and assessment. This paper focuses on the development of security checklists for use with student laboratory work. Our plan is a work in progress; initial implementation began spring, 2007, with preliminary results available in June 2007. We are actively seeking partnership and collaboration opportunities with other universities and this paper serves as a vehicle for inviting ideas and feedback.

Index Terms: Security Education, Computer Science Curriculum, Information Security Curriculum Development, Risk Management

I. INTRODUCTION

Industry has recognized that creating secure systems requires incorporating security concepts throughout the software development lifecycle. A similar effort is required in education, integrating security best practices and risk management throughout the curriculum. Currently, education addresses security primarily through tracks and security courses, after the students have developed coding techniques. In an earlier paper [1], we outline a prototype plan to integrate security into the computer science/computer information systems curriculum at Towson University. After offering a dedicated track in computer security [2][3][4] for four years, we firmly believe that a threaded approach is necessary to thoroughly prepare our students for the 21st century workforce [5][6][7][8][9][10]. We have developed a plan to integrate security touchpoints and risk analysis into the core courses and two upper level courses from each major. This model includes customized labs to enforce secure coding mantras and a black hat/white hat approach to teach students to identify and mitigate risks [1]. Key to our plan is the use of security checklists and scorecards. Checklists provide a quantifiable list of security criteria to aid in writing secure code and further reinforce the security principles. Additionally, scorecards and checklists provide a consistent means of evaluation and assessment. This paper focuses on the development of security checklists for use with student laboratory work.

This plan is a work in progress; initial implementation began spring, 2007. This paper serves as a vehicle for inviting ideas and feedback. Additionally, we are actively seeking partnership and collaboration opportunities with other universities. All materials and data produced during this study will be shared and available for educational use.

II. PROJECT OVERVIEW

Currently, most security efforts at the undergraduate level are in the form of specialized security classes, which address particular topics, or security tracks, which are effective at training security experts. Both approaches address only a subset of the students and occur after students have established a foundation of coding techniques. In other words, we are doing too little, too late. Increasingly, security education experts are recognizing the need to integrate or thread security principles across the entire computer science curriculum, beginning with the foundation courses [5][6][7][8][9][10][11]. To date, there has been little actual implementation of the threaded approach which provides security education for all computing students and establishes secure coding techniques from the onset.

Towson University is a pioneer in security education. We are a designated National Center of Academic Excellence (CAE) in Information Security and Assurance Education, and, since 2002, are one of a handful of universities that offers a specialized security track for Computer Science (CS) majors. Results from the track have been positive; it has attracted highly qualified students and initial industry feedback has indicated that the graduates are well prepared. However, minority representation is poor and, to date, there are no females enrolled in the track. 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. Consequently, we are convinced

that the next step is to infuse security principles and risk analysis throughout our entire undergraduate CS and Computer Information Systems (CIS) curriculum. Our objective is to broaden the scope of our security initiative to include all computing majors and to introduce security concepts from the first course.

Beginning spring 2007, we are implementing a threaded model that begins with the core courses and an additional two required upper level courses from each major. The courses are detailed in the following section. The cornerstones to our approach include incorporating security into the software development lifecycle, promoting software security touchpoints intrinsic to robust and secure coding, and integrating risk analysis into existing curriculum. Exposure to these concepts will begin with the foundation courses and be reinforced throughout the course of study.

In order for integration to be feasible, the impact on the already overextended CS and CIS curriculum must be as seamless as possible. The approach we propose will use existing opportunities in the curriculum. We have created lab materials for faculty to incorporate within their courses [1]. To complement the labs, we are developing security checklists to help student assess the security of their code and to further enforce the security principles. The checklists will be re-used as an assessment tool or scorecard by a grader. This paper focuses on the development of the security checklists.

A. Project Curriculum

The core courses for the Computer and Information Sciences Department are as follows:

COSC 175 – General Computer Science (CS0) logic course taught in pseudocode

COSC236 – Introduction to Computer Science I (CS1) introduces imperative programming using C++

COSC237 – Introduction to Computer Science II (CS2) covers object oriented concepts using C++

CS1 and CS2 are required of the CS and CIS major, with many students also taking CS0 to meet the programming prerequisite in CS1. These courses serve as the springboard for all other courses in these majors. The following courses have also been identified for security integration:

- COSC455 – Programming Languages: Design and Implementation (PROG LANGS)
- CIS 379 – Systems Analysis and Design (SYS ANAL)
- COSC 457 – Database Management Systems (CS-DB)
- CIS 458 – Organizational Database Management (CIS-DB)

III. CHECKLISTS AND SCORECARDS

One web definition [12] describes a checklist as a **"**list of items to be noted, checked, or remembered." Despite the dubious reference, the definition is particularly relevant to our application. Our intention in creating security checklists is to enumerate, or "note", a list of security items which students will use to "check" their code with the ultimate objective being that students internalize or "remember" key security principles. Bishop enumerates the role of checklists in education, as a reminder list and set of quantifiable items that a student must satisfy, and as a measurement tool used by a grader [13][14]. This paper distinguishes between checklists and scorecards, borrowing from [15]. A checklist is completed by a student as a self-test; a scorecard is completed by an independent reviewer or grader. Scorecards provide a standard means for evaluating projects [15].

The basis for our checklists and scorecards are well-established security mantras that guide software security [1]. Bishop [14] states, "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."

Checklists have advantages and disadvantages. On the positive side, 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 [15]. Designing fully secure code is psychologically difficult [15], "why would anyone do that?", "you want me to break my own code?" Checklists can help to find potential vulnerabilities. Finally, checklists reinforce security principles and help the student internalize key concepts. On the negative side, there can be an over-reliance on an enumerated list, leading to the idea that once the checklist is complete, the specific task is complete. The checklist itself may be poorly developed or incomplete. Additionally, the user, in our case, a programming student hurrying to submit a project, may mark off items inaccurately. Effective use of security checklists requires training, repetition, and feedback.

Checklists are used in many applications to reduce the likelihood of human error, most notably in aviation safety. While pre-flight checklists have been considered a key method in improving airline safety, checklists are

increasingly used in software assurance. Bishop advocates the use of principle-based checklists in industry [16] and in education [14].

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. Ultimately, the purpose of the checklist is to help students master security principles.

A. Sample Checklists

Following is a partial list of checklists for the core courses: CS0, CS1, and CS2. The format of the CS0 checklist is slightly different from the subsequent checklists for CS1 and CS2, due to the inexperience of CS0 students. The CS0 checklists require students to explicitly list variables and types, etc., before checking off items. These examples serve as a starting point towards the development of a complete set of checklists for the core courses. The checklists included below are necessarily compact due to space limitations. Actual checklists distributed to students will include:

- sample code – errors to look for
- examples – correct ways of writing code
- security mantras – a list of principles that form the basis for the checklist, for example: "All Input is Evil!"

Additionally, a comprehensive checklist, for use with any project, will be developed.

i. 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 [17]. Integer overflows are difficult to detect and attacks of this type are increasing [18]. While avoiding signed numbers simplifies the secure coding process [18], in our checklists, we emphasize strong and appropriate typing. The goal is to instill secure coding habits for future programmers.

Table 1: Security Checklist – Integer Overflow (CS0)

Security checklist			
Vulnerability		Integer Overflow	
Course		CS0	
variable	**Type OK?** (use unsigned if possible)	**Assignments OK?** (watch for overflow, especially with multiplication)	**Inputs OK?** (watch for possible overflow)

Table 2: Security Checklist – Integer Overflow (CS1)

Security checklist			
Vulnerability	Integer Overflow		
Course	CS1		
	Yes	**No**	**Unsure**
1. Is each variable appropriately typed? (hint: use unsigned if possible)			
2. Are all assignments to the variables within range? (hint: especially watch multiplication & pointers!)			
3. Are all inputs to the variables within range?			
4. Are all parameters passed to functions within range?			
5. Is the parameter pass by reference?			
Shaded area indicated this is a high security risk!			

Table 3: Security Checklist – Integer Overflow (CS2)

Security checklist			
Vulnerability	Integer Overflow		
Course	CS2		
	Yes	**No**	**Unsure**
1. Is each variable appropriately typed? (hint: use unsigned if possible)			
2. Are all assignments to the variables within range? (hint: especially watch multiplication!)			
3. Are all inputs to the variables within range?			
4. Are all parameters passed to functions within range?			
5. Is the variable used for memory allocation?			
6. Is the variable used for an array subscript?			
Shaded area indicated this is a high security risk!			

ii. Buffer Overflow

Considered the "nuclear bomb" of the software industry [19], 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 [19]. The infamous buffer overflow gained public notoriety in 1988, yet still accounts for up to 50% of all software vulnerabilities [20]. While any discussion of C and C++ includes warnings about buffer overflow, the security implications of the buffer overflow are demonstrated following a discussion of program run-time environment and memory layout.

Table 4: Security Checklist – Buffer Overflow (CS0)

Security checklist				
Vulnerability		Buffer Overflow		
Course		CS0		
Array name	**Upper bound**	**Assignments OK?** (watch for assignment beyond upper bound)	**Inputs OK?** (watch for possible overflow of index)	**Loops OK?** (watch for possible over-flow of index)

Table 5: Security Checklist – Buffer Overflow (CS1)

Security checklist			
Vulnerability	Buffer Overflow		
Course	CS1		
	Yes	**No**	**Unsure**
1. Are loops operating on arrays within range (hint: watch for indices off-by-one)?			
2. Are all string functions within range(hint: do not use: gets, strcpy, sprintf)			
3. Is all input to arrays within range?			
4. Are all arrays passed to functions within range?(hint: check if input is NULL, input strings are terminated with NULL, check lengths)			

Table 6: Security Checklist – Buffer Overflow (CS2)

Security checklist			
Vulnerability	Buffer Overflow		
Course	CS2		
	Yes	**No**	**Unsure**
1. Are loops operating on arrays within range (hint: watch for indices off-by-one)?			
2. Are all string functions within range(hint: do not use: gets, strcpy, sprintf)			
3. Is all input to arrays within range?			
4. Are all arrays passed to functions within range (hint: check if input is NULL, input strings are terminated with NULL, check lengths) ?			
5. Are 1-4 checked for dynamic data allocation also?			
6. Do operations to dynamic arrays stay within range?			
7. Are arrays close to pointers? (Hint: remember the return address of a function s a pointer)			
Shaded area indicated this is a high security risk!			

iii. Input Validation

Input validation encompasses several security mantras:

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

While programmers' focus has traditionally been on making things work and making things possible, secure coding requires a shift of focus to making invalid input impossible. Writing code that rejects incorrect data, a potentially infinite set, is both costly and insecure. Secure coding is accomplished by accepting only valid data and rejecting anything else. In other words, deny by default.

Table 7: Security Checklist – Input Validation (CS1)

Security checklist			
Vulnerability		Input validation	
Course		CS0	
Variable input	**Type OK?**	**Values OK?**	**Only accept valid?**

Table 8: Security Checklist – Input Validation (CS1)

Security checklist			
Vulnerability	Input validation		
Course	CS1		
	Yes	**No**	**Unsure**
1. Is the value input checked for the correct type?			
2. Is the value input checked for within range?			
3. Is the value input checked not to overflow a buffer?			
4. Do input statements only accept valid data and reject everything else?			
5. Do functions check for valid input?			
6. Are input values for array indices?			
Shaded area indicated this is a high security risk!			

Table 9: Security Checklist – Input Validation (CS2)

Security checklist			
Vulnerability	Input validation		
Course	CS2		
	Yes	**No**	**Unsure**
1. Is the value input checked for the correct type?			
2. Is the value input checked for within range?			
3. Is the value input checked not to overflow a buffer?			
4. Do input statements only accept valid data and reject everything else?			
5. Do functions check for valid input?			
6. Are input values for dynamic memory allocation?			
Shaded area indicated this is a high security risk!			

IV. EVALUATION

One of the most challenging and criticized component of any new approach is the assessment and evaluation of the project outcome. There are three components to our evaluation:

- Awareness assessment – Pre and post-tests will be developed at each course level to assess students' security literacy.
- Security checklists – Students complete for each lab.
- Security scorecards – Graders complete scorecards and students receive a security score for each lab.

Data from all three components will be compiled and analyzed across the class sections. Other evaluation tools to be considered are risk checklists and scorecards – to track risk awareness and the risk mitigation rates; faculty surveys, exit surveys, and alumni surveys; and security maturity scores – for term projects in upper level courses such as SYS ANAL and DBASE. Metrics will be developed to apply the results of the assessments to calculate a security fluency score.

V. CONCLUSION

This paper argues that a threaded approach to security is the only model that adequately prepares future software professionals. A key idea in implementation of this model is the use of checklists based on security principles. Security checklists reinforce key security concepts and help students design and create secure code. Additionally, the use of scorecards and checklists provides a standard means of evaluation and assessment.

This paper is a work in progress. We plan to implement our first pilot program in spring 2007 and have preliminary results by summer 2007. Ultimately, we intend to infuse security throughout our entire computing curriculum. Our goal is to provide all computer and information science undergraduates with the skills necessary to design and code secure systems.

VI. REFERENCES

[1] B. Taylor and S. Azadegan, Threading Secure Coding Principles and Risk Analysis into the Undergraduate Computer Science and Information Systems Curriculum, *INFOSECCD*, 2006.
[2] S. Azadegan, M. Lavine, M. O'Leary, A. Wijesinha, and M. Zimand, A dedicated undergraduate track in computer security education. In *Security Education and Critical Infrastructures*, 2003.
[3] S. Azadegan, M. Lavine, M. O'Leary, A. Wijesinha, and M. Zimand, M. An undergraduate track in computer security. In *Proceedings of the 8th Annual Conference on innovation and Technology in Computer Science Education*, Greece, 2003.
[4] S. Azadegan, M. Lavine, M. O'Leary, A. Wijesinha, and M. Zimand. Undergraduate Computer Security Education: A Report on our Experiences & Learning. *Proceedings of Seventh Workshop on Education in Computer Security*, Monterey, CA, 2006.

[5] J. Davis and M. Dark. Teaching Students to Design Secure Systems, *IEE Security and Privacy*, Vol. 1, Num. 2, March 2003.
[6]C. E. Irvine, S. Chin and D. Frincke. Integrating Security into the Curriculum, *IEEE Computer*, pp. 25-30., Dec. 1998
[7]L.F. Perrone, M. Aburdene, and X. Meng. Approaches to undergraduate instruction in computer security, *Proceedings of the American Society for Engineering Education Annual Conference and Exhibition*, ASEE 2005.
[8] R. Vaughn, Jr., Application of security to the computing science classroom, *Proceedings of the thirty-first SIGCSE technical symposium on Computer science education*, p.90-94, , Austin, TX, March, 2000.
[9] G. White and G. Nordstrom. Security across the curriculum: using computer security to teach computer science principles. *Proc. 19th Nat'l Information Systems Security Conf.*, Baltimore, MD, 1996.
[10] A. Yasinac and J.T. McDonald, Foundations for Security Awareness Curriculum, *Proceedings of the 39th Hawaii International Conference in System Sciences*, 2006.
[11] S. Azadegan, M. O'Leary. Incorporating Security Concepts into First Course. *International Workshop on Informatics Education: Bridging the University/Industry Gap*, Santiago, Chile, 2006.
[12] Answers.com
http://www.answers.com/topic/checklist
[13] M. Bishop and D. Frincke, *Teaching Secure Programming*, IEEE Security and Privacy 3(5) pp. 54-56, Sep, 2005.
[14] M. Bishop, Teaching Assurance Using Checklists, *Seventh Workshop on Education in Computer Security*, Monterey, CA, 2006.
[15] M Graff, K. van Wyck. *Secure Coding: Principles and Practices*, O'Reilly, Sebastopol, CA, 2003.
[16] D. Gilliam, T. Wolfe, J. Sherif, and M. Bishop, Software Security Checklist for the Software Life Cycle, *Proceedings of the Twelfth IEEE International Workshops on Enabling Technologies:Infrastructures for Collaborative Enterprises*, 2003.
[17] R. Seacord, Secure Coding in C and C++ Of Strings, and Integers, *IEEE Security and Privacy*, 2006.
[18]M. Howard and D. LeBlanc. *Writing Secure Code*, Microsoft Press, Redmund, WA, 2003.
[19]G. Hoglund, G. McGraw. *Exploiting Software: How to Break Code*, Addison-Wesley, Boston, 2004.
[20] J. Viega and G. McGraw. *Building Secure Software*, Addison-Wesley, Boston, 2002.