TRAC: AN APPROACH TO TEACHING SECURITY-AWARE PROGRAMMING IN UNDERGRADUATE COMPUTER SCIENCE COURSES

Rochelle Elva

Department of Mathematics and Computer Science,
Rollins College, Florida, USA

ABSTRACT

The unfortunate list of software failures, attacks, and other software disasters has made it apparent that software engineers need to produce reliable code. The Department of Homeland Security reports that 90% of software exploits are due to vulnerabilities resulting from defects in code. These defects are easy to exploit. They are potentially dangerous as they create software vulnerabilities that allow hackers to attack software, preventing it from working or compromising sensitive data. Thus, these defects need to be addressed as part of any effort to secure software. An effective strategy for addressing security-related code defects is to use defensive programming methods like security-aware programming. This paper presents TRAC, an approach to teaching security-aware programming. The acronym stands for Teach, Revisit, Apply and Challenge. It also describes the implementation of the approach and the results of a small case study (n = 21), in a senior-level elective course.

KEYWORDS

Security-Aware Programming, Secure Coding, Software Security, Teaching Secure Coding.

1. INTRODUCTION

In our modern world, every facet of our lives from the most mundane activities in our homes to the more complex activities like medical care, all rely to some extent on software. These systems utilize software to manage a wide range of applications and services. They handle sensitive data including personal and financial information and control processes that are at times life-threatening. As a result of this strong dependency on software, even a minor security breach can have a ripple effect resulting in tremendous damage that cannot be contained or localized. The unfortunate list of software failures, attacks, and other software disasters has made it very apparent that as software engineers, we need to produce reliable code- that is secure code.

The Department of Homeland Security reports that 90% of software exploits are due to vulnerabilities that result from defects in code [1]. These defects are easy to locate and exploit. They are potentially dangerous as they create software vulnerabilities allowing hackers to attack software, rendering it non-operational and/or compromising sensitive data [2]. Thus, any effort to secure software must include the management of these code defects.

David C. Wyld et al. (Eds): FCST, CMIT, SE, SIPM, SAIM, SNLP - 2022
pp. 111-129, 2022. CS & IT - CSCP 2022 DOI: 10.5121/csit.2022.120809

An effective strategy for addressing security-related code defects is to use defensive programming methods [3], such as security-aware programming. These programming methods are designed to build reliable systems. They achieve their objective by incorporating security considerations into the code design process so that the software produced is free from flaws that will make it vulnerable to attack. Such systems can be trusted to perform reliably, even under unexpected conditions. From a coding standpoint, software developed defensively will be free of security-related defects such as buffer overflow errors, null pointer deference, and improper input and output validation errors.

Incorporating secure coding instruction in the undergraduate curriculum would provide our students with the ability to code securely. This is a necessary skill to prepare them for their careers [4]. Currently, there is often a knowledge gap between the coding demands of the industry and the ability of graduating students to write robust code. To bridge this gap, many high-tech companies must provide security training for new hires [5]. Security awareness training discussed by Banerjee and Panday in [6], is just one approach that some companies use. The use of security-aware coding instruction - like TRAC, in our university programs, would fill this knowledge gap and provide graduates with skills that will make them immediately marketable.

This paper presents TRAC, an approach to teaching security-aware programming. The acronym stands for Teach, Revisit, Apply and Challenge. It is a four-step approach devised to facilitate the development of mastery in writing secure code. Our approach is implemented as a module across multiple existing, core and elective courses in the computer science curriculum. This paper describes the implementation of our approach to security-aware programming and presents the results of a small case study, used as a pilot test.

2. BACKGROUND AND REVIEW OF LITERATURE

In this section, we will define the terms that will be used in this paper, explain our rationale for teaching security-aware programming, and provide a review of related work in software security - particularly software security education. We will also discuss some of the obstacles that contribute to the lack of security-aware programming instruction in the undergraduate computer science curriculum.

2.1. What is Security-Aware Programming?

To define security-aware programming, we first must define two fundamental concepts: code defects and security-related software vulnerabilities. Code defects refer to errors in code. We will focus on logical errors, not syntax errors. We assume that the target student group is capable of writing basic code that compiles. Security-related software vulnerabilities are weaknesses in software, that stem from code defects that can be exploited. Their presence in code, therefore, makes it less secure. For the purposes of this paper, security-aware programming is defined simply as coding securely. This is the skilled practice of designing and writing code so that the final product is free of defects, that could lead to security-related vulnerabilities. As a result, the code (and software produced) is robust and reliable. Like any other skill, security-aware programming is developed and refined through repeated practice. We use the contextual approach to learning presented in [7]. This involves learning to identify the code defects to be avoided and engaging in the application of the relevant strategies to prevent them in a variety of situations. Thus security-aware programming involves content from the cross-cutting bodies of knowledge including software engineering and the fundamentals of software and program development [3, 8]. The actual implementation of security-aware programming would incorporate all the skills required to build robust code. These skills would include identifying test cases that provide full

coverage of the code and testing code throughout the development process including code review. In related work, the concept of security-aware programming is referred to as defensive programming [3, 4, 9], robust programming, secure programming [3, 4], having a security mindset [10], and coding using risk management [11].

According to the report of the 2008 Secure Coding Workshop, while coding security features in code is the job of only a few security specialists, security-aware programming is the responsibility of every programmer. They continue with the claim that security-aware programming is a requisite for meeting the security requirements of code [4].

2.2. The Rationale for Teaching Security-Aware Programming

Gary McGraw states that external approaches to securing software are nowhere as effective as designing software that is secure in the first place [12]. The Department of Homeland Security cites The Software Engineering Institute as reporting that 90% of software exploits are due to vulnerabilities that result from defects in code [1]. The presence of these defects needs to be addressed since they are easy to identify and exploit during attacks such as DOS (denial of service) [2]. The failure to practise defensive or secure coding has been identified as the cause of many of the defects in software [3, 13]. As a result, there has been some discussion and research on the value of teaching security-aware programming and how this skill can be incorporated into the undergraduate computer science curriculum. At the 2008 Secure Coding Workshop, industry representatives lamented the time and other resources needed to train new employees in the skills required to write secure code. They also advised that students should enter the job market already skilled in secure software development [4]. In 2010, the Summit on Education in Secure Software was convened to identify the specifics of the security content that students need to learn and to suggest effective teaching strategies [14]. Then the 2013 Computer Science Curricula added Security as part of the Computer Science Body of Knowledge in undergraduate Computer Science programs. Nine core hours were allocated for the security knowledge areas. This included fundamental concepts in security, design principles, and defensive programming [9]. It is evident from all of these efforts, that at all levels, stakeholders agree that security should be an integral part of every Computer Science program. However, for many of the reasons stated in Section 2.3 security-aware programming is totally absent, left to chance, or taught in a very limited way in many of our undergraduate Computer Science programs [15].

Our review of literature strongly supports the idea that security-aware programming should be an essential component of computer science education [2– 4, 6, 8, 9, 12, 16–18]. This is because security is a functional requirement for all software in our modern social environment. Most times students discover how to make their code robust through a process of trial and error, but the topic is hardly ever discussed in undergraduate courses, particularly at the introductory level. The TRAC approach to security-aware programming being proposed in this paper is designed specifically to provide multiple opportunities for students to develop the skill of writing robust code. A primary difference between the proposed approach and current practice is that the learning of secure coding skills is facilitated by actual curriculum design, instead of just being left to chance, as is often the case currently.

Another benefit of teaching security-aware programming is the positive impact on students' careers. Learning good secure coding habits includes understanding the value of test coverage to evaluate the efficacy of code. This is important since a large proportion of the coding aspects of the technical interview evaluate just that. Unfortunately, even students who are good programmers, often fail this aspect of the interview because they lack the skill of writing code that is fully robust. The security-aware code development paradigm will provide opportunities to develop the requisite skills, thus making students more marketable [4].

The vast amount of software failures and disasters has made it very apparent that the production of reliable code is a fundamental requirement and ethical responsibility of every software engineer. By its very definition, secure code is reliable code. Therefore, it is our responsibility as educators to teach security as part of software development. In addition, students who learn to code securely from the onset are more likely to continue this practice in their careers as this would have become second nature to them after years of repeated practice. It is also easier to teach individuals to master a skill by teaching the correct technique the first time, rather than attempting to correct deficiencies from years of bad practice [4].

The teaching of security-aware programming has taken one of three forms [8]:

- single concentrated course [19, 20]
- threading or integration in courses already existing in the curriculum [10, 21–25]
- concentration/track in a degree program [26]

Deciding on which approach to use is important. There are arguments for each approach [23]. For example, having a separate security class facilitates focus and depth of learning, and tends to be very effective since it is taught by faculty who are invested in the topic [3]. The integrated approach also has its advantages since there are multiple opportunities for concept formation through repetition. This approach also has high impact value since more students would have the opportunity to be exposed to the security content with little disruption of the curriculum. However, it would involve 'buy-in' from all faculty teaching the classes with software security content. The third option, teaching security awareness as a concentration/track, has the disadvantage of needing specially trained faculty and the possibility of low impact, since students may not select the concentration/track. However, this approach would provide the benefit of depth of learning for the students who do select the concentration/track.

The TRAC approach to teaching security programming is a threaded approach that provides several advantages and addresses some of the issues just discussed. This approach allows faculty who appreciate the value of security education to include the approach across their classes. The use of TRAC does not alter the course schedules and provides learning opportunities for students across multiple courses. This will therefore provide impact across the curriculum even if only one faculty member 'buys in'.

2.3. Obstacles to Teaching Security-Aware Programming

Although the Computer Science Curricula 2013 has recommended that security be infused into the computer science curriculum at all levels [9], almost 10 years later, this recommendation has not been implemented in several programs. This can be attributed to two primary reasons: perceived lack of resources and failure to believe in the merit of teaching security-aware coding. We will now discuss some barriers to the teaching of security-aware programming that we have identified through our research.

The issue of lack of resources has two main components: faculty, and curriculum bandwidth. Several institutions state that they do not have faculty with software security training [27]. Some faculty also complain about the absence of teaching resources [5]. Yet, several resources have been developed with materials that they can use. These include course modules and e-learning materials such as the Seed Project, OWASP WebGoat, and SWEEP project [8]. faculty are either unaware of their existence or they are not convinced of the value of the required time investment. Another problem is that many of the resources are more advanced and complex than what would be needed by faculty who are not security specialists. These resources tend to focus on Web-

based projects and cybersecurity frameworks and are therefore inappropriate for introductory-level courses. The second resource issue is a lack of both time and space in an already packed curriculum. Creating new security courses or adding content to already existing courses is considered an unnecessary burden, causing an increased workload for both faculty and students [4, 5]. In addition, some faculty believe that introductory classes should focus solely on code algorithms and syntax. The insertion of new content is considered disruptive [3].

The other issue is the perceived merit of teaching secure programming. Many faculty do not value security-aware programming as a necessary addition to the curriculum. Some believe that they are already teaching these concepts - although student feedback suggests the contrary (see Section 4.2.3). Also, while many companies would like new hires to be skilled in secure programming, they do not explicitly include this as a requirement in posted job descriptions. Consequently, some faculty and even students do not prioritize secure programming skills in the undergraduate computer science programs [4]. According to Bishop in [27], some faculty also believe that it is a myth that the security of software will be improved by teaching students to code securely. Their rationale is that this ignores the impact of other contributors to security. It is true that one cannot overlook the value of the other facets of software security, such as security infrastructure. However, as more companies begin to accept the value of security, investment in secure infrastructures will become standard and the need for secure code will remain a standard. Teaching students to code securely, will not solve every software security problem, however, it will contribute to the solution.

2.4. Related Work

Table 1 summarizes the approaches to teaching software security that we identified in our review of the literature. In the 12 articles identified, the majority used the approach of integrating security modules into already existing courses. However, only two of these spoke specifically about threading these modules across multiple courses at different levels in the undergraduate program [18, 23]. In two of the articles, a single specialized course was used to introduce an in-depth coverage of software security [19, 20]. While all the articles suggested ways to incorporate the learning of software security principles into the undergraduate curriculum, their approach and focus were different. Some approached the teaching of software security as secure software design, while others used defensive coding. Numbers 1-4 and 9 in Table 1, used the secure software design approach, adding a level of security to system development life cycle (SSDLC). Numbers 5-7 and 12 in Table 1, used the defensive coding approach. In four of the articles, the focus was on teaching security as soon as possible, so the target group was the introductory computer science classes. These are represented by numbers 5-8 in Table 1. Two articles focused on specific software security issues: secure mobile computing [28] and digital forensics [29]. In two of the articles, the focus was to educate faculty. This served both to train the faculty and to provide resources that they could reuse in their courses [5, 23].

Table 1. Summary of the approaches to teaching software security that we identified in our review of literature

Teaching Format	#	Approach	Focus	Source
Single Course	1	Software Design	Secure Software Design	[19]
	2	Secure Software Design SSDLC	Senior-level security course	[20]
Integration in existing course/s	3	Software Design	Data Structures Software Design	[21]
	4	Software Design (SS-DLC)	Intro to Java Software Engineering	[22]
	5	Defensive coding	Introductory Classes	[10]
	6	Defensive coding	Introductory Computer Science classes	[16]
	7	Defensive coding	Introductory Computer Science classes	[24]
	8	General Security Topics - digital forensics	Introductory course	[29]
	9	Secure Software Design for Mobile Apps	Courses in Mobile app development	[28]
	10	Resources & Tools	Faculty training workshop	[5]
Integration in existing course/s and Threaded throughout the curriculum	11	Secure Software Design SSDLC using software case studies	Faculty training workshop	[23]
	12	Secure Software Design and defensive programming	6 courses (including introductory core courses) throughout the curriculum	[18]

3. TRAC APPROACH TO TEACHING SECURITY-AWARE PROGRAMMING

In this section, we present a detailed description of the TRAC approach to developing the skill of security-aware programming. We also identify the set of software defects that will be the focus of our instruction.

TRAC is an acronym for Teach, Revisit, Apply and Challenge.

Our approach is intended for use as a module in any code-based computer science course. To overcome some of the obstacles to teaching secure coding (discussed in Section 2.3), our approach works with existing courses. The techniques used can be implemented by faculty without specific training in software security. It can also be used across multiple courses, to facilitate incremental skill development through repeated practice in a variety of contexts. This is supported by Ambrose et al. in their book on how to learn. They claim that it takes at least 21 repetitions of the correct way to perform a skill before it becomes a habit.

3.1. Implementation of TRAC

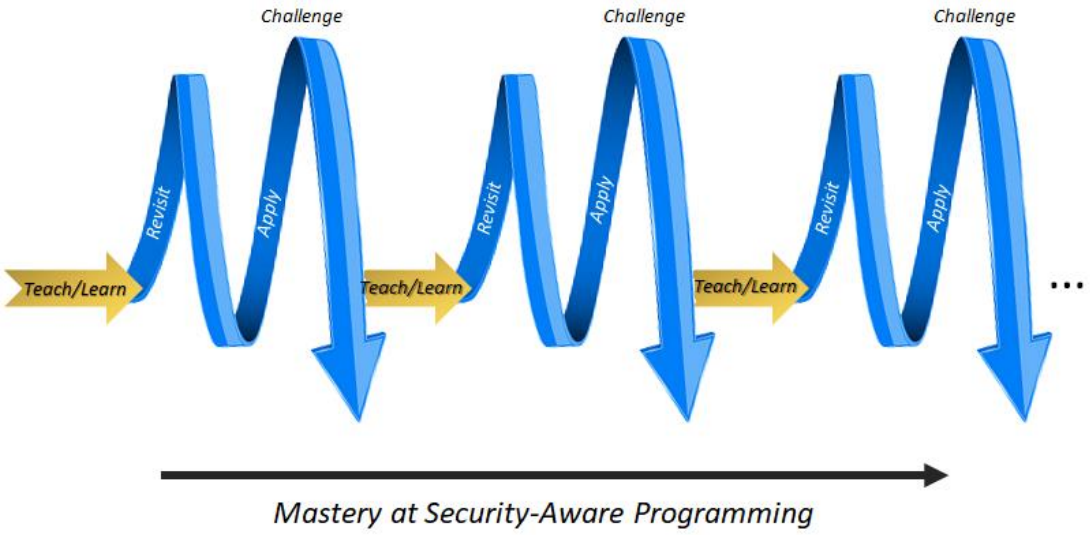

Figure 1. The TRAC Approach to Teaching Security-Aware Programming

Figure 1 illustrates our thinking on how mastery of security-aware programming skills can be developed using the TRAC approach. Learning will occur in repeated phases of teaching/learning, revisiting written code to evaluate its level of security, applying the knowledge of security-related code defects to adjust code to improve security, and challenging each other during code review. The ultimate challenge is the self-challenge - to be able to write code needing little or no security improvements.

We selected a spiral and not a cycle to represent our learning process because while the phases are repeated, the starting point for repetitions is not the same. Each experience improves the level of skill development, thus moving the learner further along the learning continuum, and closer to mastery.

3.1.1. Teaching

During the teaching stage, students will be instructed using the content described in Section 3.2.1. As a result of these experiences, students should be able to name the defects described in Table 2. They should also be able to identify these code defects in new code examples. Students will also use traditional code design tools like activity diagrams and class diagrams to identify code interface points and the data traveling across those interface points. They will learn to use these to determine potential data security issues.

3.1.2. Revisiting Stage

Once students have developed proficiency in the identification of security flaws, they will be invited to examine their previous coding assignments to find the unchecked security-related code defects that made their programs vulnerable. They will then select examples that they feel comfortable sharing with the class. The class will then discuss the presentations, identifying the most common defects found and any others that might have been missed by the presenters.

3.1.3. Application Stage

At the beginning of the module, the application stage will involve two types of activities. The first will be to apply the knowledge of security-aware programming to correct the security flaws identified in a previous assignment submission. The second will be to use the skills learned, in a completely new assignment to write code that is even more secure than their previous work. As students master the skill of security-aware programming, they will automatically use the strategies for the avoidance of code defects, to produce secure code that is error free. Through practice, writing secure code will become more natural and second nature.

3.1.4. Challenge Stage

During the challenge stage, students are given coding problems that they will solve in groups. As they work on their solutions, they will make a list of the security checks that they have considered. All team members will contribute to the final deliverable. Teams will then challenge each other to break the code created. As students progress through this process, the challenge stage will evolve into formal code reviews. This will prepare students for the code review process that is a common practice in the industry. Through the activities of this stage, students will learn how to prepare their code for review and how to critically review code prepared by their peers.

3.2. The Learning Goals of TRAC

The TRAC approach is designed to create opportunities for students to acquire the skill of writing robust code through awareness of security concerns associated with software. This goal of the approach is expressed in the following two learning outcomes: as a result of using the TRAC approach, students should:

- acquire code security knowledge
- develop mastery in the skill of security-aware programming

3.2.1. Acquisition of Knowledge

We believe that code security knowledge involves both the learning of software security content and an understanding of the contextual relevance of coding securely. Thus, facilitating the acquisition of code security knowledge instruction in TRAC begins with building a rationale for secure programming. Teaching security-aware programming using TRAC, fits into the Information Assurance and Security knowledge areas, added to the computer science curriculum in 2013 [9]. This knowledge area has the following five learning outcomes:

1. Analysis of the trade-offs of balancing security properties
2. A description of risks, threats, and vulnerabilities and how these relate to security attacks
3. Understanding the concepts of trust and trustworthiness in terms of software
4. OS SEcurity and Network Security
5. HCI

The TRAC approach addresses the first three of these learning outcomes. This is expressed in our teaching/learning objectives that students should be able to:

- explain the rationale for security-aware programming
- list and identify common code defects that are security risks
- apply design and coding principles of defensive programming to mitigate security-related code defects.

Table 2. Common Code Weaknesses Adapted From The Common Code Weakness Enumeration

CWE ID	Weak-ness	Highest Position	Description	Likelihood of Exploit:	Negative effects	Mitigation Strategies
787	Out of Bounds Write	1	Code writing data to a position before or after the memory location of a given buffer	high	Code crash, DOS, modifying memory	input validation of write parameters
125	Out of Bounds Read	3	Code reading data from a position before or after the memory location of a given buffer	high	Code crash, DOS, modifying memory	input validation including calculations producing length parameters
20	Improper Input Validation	3	Code receives and uses data without setting in place checks and balances that the values received are legitimate	high	Code crash, DOS, entire system hijacked by ransomware	Adopt a non-trust policy treat all input as untrustworthy analyze code and design for possible areas of insecure input and validate input
190	Integer Overflow	8	the results of a calculation that produces a value larger than an integer; code attempts to store that value as an integer	high	Buffer overflow, Code crash or infinite loop	input validation and validation of the result of integer calculations; using unsigned integers
129	Improper Validation of Array Index	14	Code either fails to validate array index values leading to code errors including out-of-bounds reads and writes	high	Code crash, DOS, unexpected code behavior, memory corruption, out-of-bounds read, out-of-bounds write	Adopt a no-trust policy, data validation including input validation for all data used as array index
476	Null Pointer Deference	14	Code accesses or tries to use null value as if it were an actual object reference	medium	Code crash, unexpected code behavior	Validation of all object data including input validation for all data
754	Improper check for unusual orexcep-tional conditions	15	Code fails to check for edge cases and exceptional conditions in the code	medium	Code crash, DOS, unexpected code behavior	Develop test cases that provide full coverage of code, handle exceptions locally instead of throwing them to other parts of code, anticipate error conditions

						and program code to exit elegantly
835	Infinite Loop	26	Code gets into a loop and does not have a condition to get out	no known attack pattern	Code crash due to consumption of memory, DOS	Check that all loop terminating conditions can be reached; input validation for loops managed by input data
532	Insertion of sensitive data in log file	33	As part of error handling, code unwittingly writes security-sensitive data such as code structure, file names and format to log file.	high	Attackers gaining access of log file have access to sensitive data and an unprotected path to security data	Careful selection of messages sent to log files; Sensitive error log messages used during code development and testing should be erased when no longer needed

We subscribe to the opinion cited in [3, 30] that acquiring the relevant knowledge will affect what the students observe and how they use these observations to solve new problems. For this reason, in the teaching component of our approach, we provide content that will help students to understand why they should care about code security. To establish this context, we review notorious major software failures and discuss and analyze reports from multiple sources including the Department of Homeland Security (DHS), Software Engineering Institute (SEI), and reputable new reports on current events explaining the impact of code defects.

To study code defects, we selected from an established list of verified software code weaknesses. Our source was the Top 25 Common Errors Enumeration from CWE/SANS [2]. This source ranks software defects based on their prevalence, and impact on code security. The most common and harmful defects are found at the top of the list. We examined lists from 2010 to 2021.

Several of the 25 top code weaknesses listed were not relevant to our target audience. For example, many of those listed focused on web-based software applications. We, therefore, filtered the list, keeping application-independent flaws that would be contextually relevant to most students in the computer science undergraduate program. Our final selection was the set of nine code defects shown in Table 2.

For each defect in the Table, we provide a description of the defect, an explanation of the negative impact that it can have on code, and the likelihood that this flaw would be exploited. We also mapped each defect to a list of strategies that can be used to mitigate its occurrence in code. To create a discussion-point on the relative significance of the defects selected, the highest position, held in the top 25, is presented in the Table. This highest position refers to the highest-ranking that each specific defect, ever occupied in the list of top 25 common errors, during the time period that we examined (2010 - 2021). The first eight defects in Table 2 were listed among the top 25 weaknesses at some time during our research time period. The ninth defect in our list (insertion of sensitive data in log files), was never in the top 25, it was listed in the top 35 in 2019. However, we decided to include this error handling defect for the following three reasons: there was a high likelihood that the defect would be exploited; it was a good teaching tool, and it would be relatable to students.

3.2.2. Skill Development

Our primary objective is to take learners from the novice level of security-aware programming to the level of proficiency - as experts. Borrowing from the developmental learning approach in [30], we evaluate mastery by focusing on two dimensions of learning - consciousness, and competence. Consciousness is the achievement of a goal through deliberate choice and focused action. Competence is the ability to perform a task with a high level of mastery or expertise.

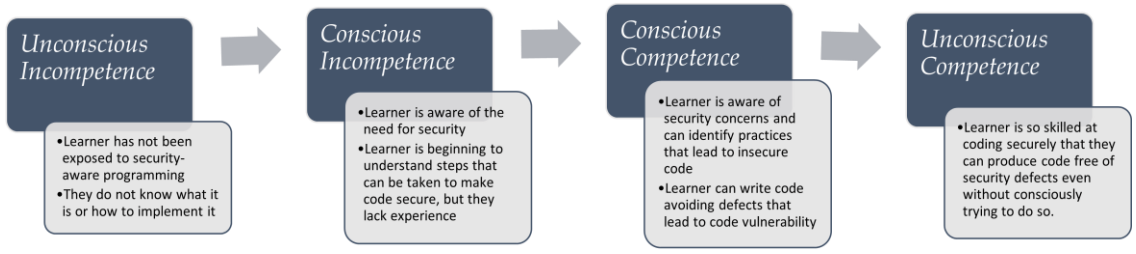

Figure 2. Progression from Novice to Expert Secure Programmer using TRAC

It is our belief that the use of TRAC will facilitate learners' maturity from being unconsciously incompetent security-aware programmers to becoming unconsciously competent security-aware programmers. The stages of transitional skill development are depicted in Figure 2.

In the first stage, the learner is unconscious of security concerns and is also incompetent at coding securely (thus unconsciously incompetent). This corresponds with the beginning of the Teaching stage of TRAC. The learner transitions to the second phase, after being schooled in the identification of security-related code defects, and their impact on the vulnerability of software. This second phase is called the consciously incompetent stage because while the learner is aware of the security concerns that need to be addressed, they have limited knowledge and ability to correct them. At this point, the learner is at the stage corresponding with the Revisiting stage of TRAC. Through practice and more learning, the learner will become both more conscious of the security concerns, and competent in the strategies used to reduce and/or avoid code defects. The learner consciously and skilfully applies their learning to produce more secure code. At this point, the learner transitions to the next phase called consciously competent. This will occur during the Application stage of TRAC. With much practice and experience, the learner will effortlessly transition to the next stage. This constitutes mastery. At this point, the practice of secure coding will be second nature. The programmer codes securely on autopilot as it were. This mastery stage is described as the unconsciously competent stage. This stage corresponds with the Challenge stage of TRAC, but it is not a static stage. The learner continues through the stages of TRAC but each time gets further along the learning continuum, and closer to mastery.

4. CASE STUDY

The TRAC approach was tested in an upper-level, elective course, on Secure Software Engineering. Security-aware programming was taught as a course module, over a period of three weeks. The Security-Aware Programming module was taught as a component of the Secure System Development Life Cycle. There were 21 students enrolled in the course: seven graduating seniors, ten juniors, and four sophomores. All students had already completed at least three computer science courses.

The pilot test was evaluated using observation of students' interactions during class, evaluating written assignments, and reviewing student feedback. There were 3 written assignments. The first 2 were identical assignments but were given 2 weeks apart. Students were asked to find a code sample that they had written and submitted for one of their previous computer science classes. They were required to analyze the code to see if there were any security-related code defects. Students were then asked to modify the code sample so that it was more secure. The third assignment was to code the backend of an automated teller machine (ATM), paying special attention to security issues. Students were asked to comment their code to indicate security concerns that they had addressed. These submissions were then presented to the class for an informal code review in the form of a class discussion. A simple assignment was selected because the group of students ranged in experience from first semester sophomores to graduating seniors. The more senior students were given the option to select their own problem and prepare a secure code solution - using the absence of security-related defects as the measure of code security.

At the end of the course, students were asked to volunteer anonymous feedback on their experiences. Data was collected from all students through an anonymous, informal survey, course evaluations, anecdotal records, informal interviews, and unsolicited conversations. No extra credit was assigned for student responses. Data collection was conducted surrounding five feedback questions (FQ). Data was also collected at the end of the semester following the course (almost four months later) to determine if the course in software security had impacted their coding habits. The latter is analyzed as FQ 6.

The following 6 feedback questions were used to obtain student feedback.

1. How would you define secure software?
2. Has your perception of software security changed during this semester? If yes how?
3. What would you say was the greatest takeaway from this course?
4. How did the course match the expectations that you had during registration?
5. Is there any area/topic covered in this course that you will use going forward? If yes, please explain
6. **four-month Check-In:** Are there any security strategies/checks that we studied last semester that you find yourself paying more attention to as you write code now?

4.1. Results from Observing Students in Class and Evaluating Assignment Submissions

From students' interactions in class, it was clear that they were engaged and enjoying the content on software security. All students participated in class discussions. One sign that students were really engaged with the content (in and out of class), is that on three occasions, different students sent an email sharing a software security story, that they had heard from current events on the news. Also, as the semester progressed and students became more confident in their ability to identify and correct security flaws, they became more willing to critique each other's work and to present their own code for critique.

In the submissions for Assignment 1, student examples were almost 100% cases of failure to validate input. By the time students submitted the second assignment, their growth in knowledge was evident. They presented more complex code with a variety of different security flaws and were able to suggest code alternatives that would improve the degree of security of the code. There was one group that struggled with finding a more complex coding example. However, after observing code presented by their peers, they were able to resubmit more complex and accurate code for the assignment. Assignment 3 was very well done. Most students solved the problem

assigned. A few groups took the challenge to develop a more complex system. They implemented a database query system, an account login validator, and a batch processor for financial transactions.

From these exercises, it was evident that students were able to identify and correct the set of security-related code flaws listed in Table 2.

4.2. Results from Students' Responses to Feedback Questions

In this section, we will discuss students' responses to each of the six feedback questions from Section 4

4.2.1. Responses to FQ 1: Student Definition of Secure Software

Figure 3 illustrates the terms used by students to describe their understanding of secure software. Most students, (between 80% and 100%), defined secure software using the ACID (availability, confidentiality, isolation, and durability) properties, associated with reliable or robust software. All students defined secure software, as the product of secure coding. A little over 70% of the students included strategies used to achieve software security in their definition. They included both reactive and proactive measures.

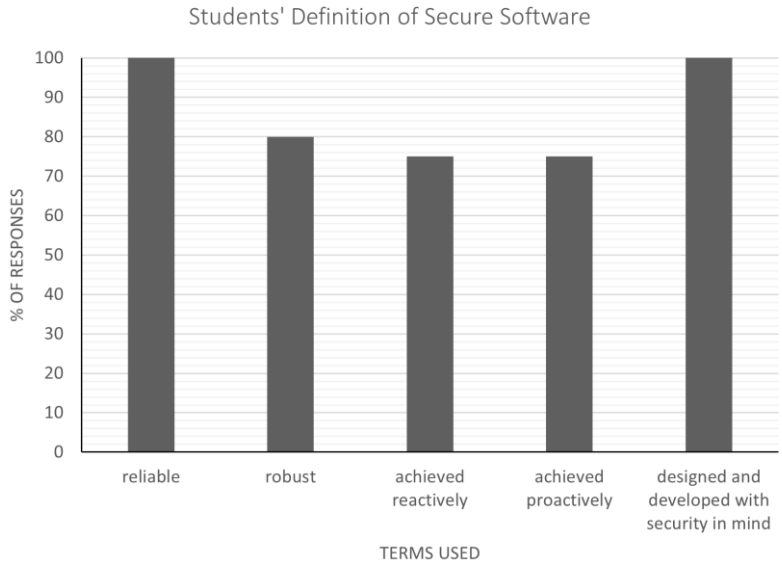

Figure 3. Graph of students' responses to Feedback Question 1

4.2.2. Responses to FQ 2: How Students' Perception of Software Security Has Evolved During the Semester

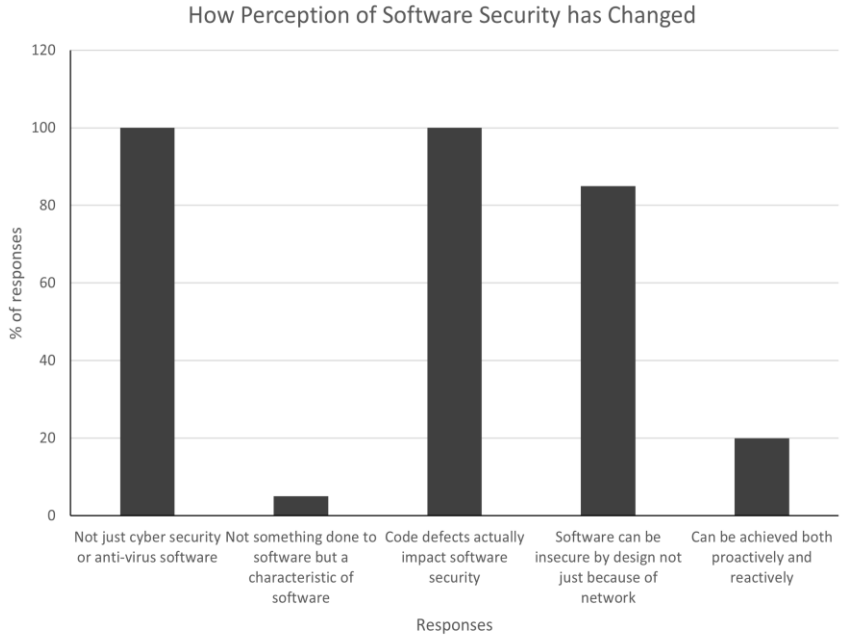

Figure 4. Graph of students' responses to Feedback Question 2

When describing how the course changed their perception of software security, students' responses generally surrounded three main themes:

- Definition of Security
- Value of Security-Aware Programming in achieving Software Security
- Their own ability to implement strategies to make their software more secure

Students explained that they had previously viewed software security as cybersecurity. At the end of the course, that perception had changed to viewing software security as a characteristic of the software and more than just cybersecurity. All students commented on a new understanding that software security can be negatively impacted by code defects. Secure coding was therefore something that they were capable of, by proactively avoiding security-related code defects. Figure 4 summarizes students' responses to this feedback question.

4.2.3. Responses to FQ 3: What Students Considered the most Significant Takeaway from the Course

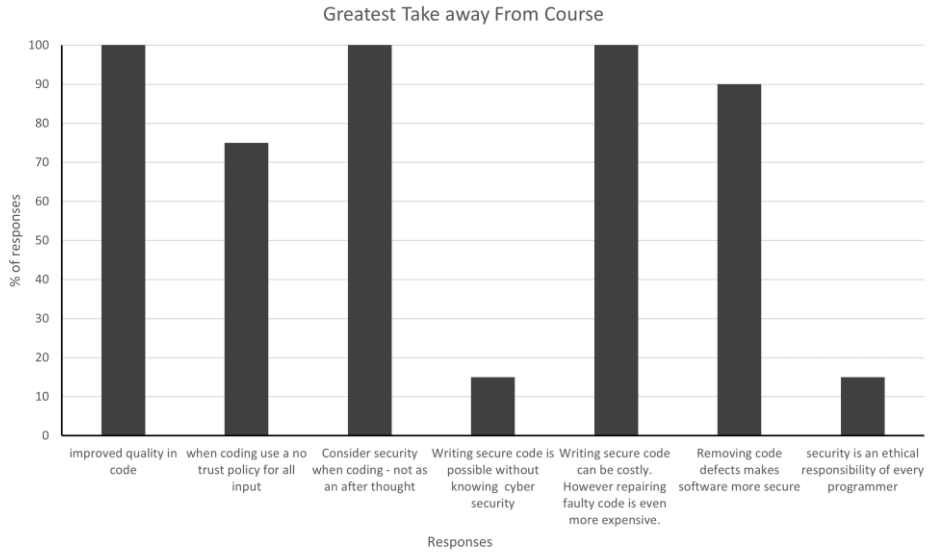

Figure 5. Graph of students' responses to Feedback Question 3

All the students considered the skill developed in the identification of software flaws as one of the greatest takeaways of the course. They all indicated that this was a skill that they were happy to have acquired as it has improved the quality of their code. Most students indicated that their code has improved because they test more extensively to ensure that all edge cases were accounted for.

Three students commented on having adopted a no-trust policy with input data, so they validate all input before using it. Students also commented on the fact that security-aware programming was a surprisingly simple, yet useful tool that they would be using for the rest of their careers. Two students commented that they have been teaching their friends how to code securely when working on collaborative projects. Two students discussed that achieving secure software may be costly in terms of time and human resources. However, the return on this investment was worth it. Another student reported that the primary takeaway was the appreciation that it is the ethical responsibility of all programmers to develop code that is reliable, and that security-aware programming is a good tool for realizing this. Figure 5 summarizes these results.

4.2.4. Responses to FQ 4: Students' Expectations of a Secure Software Engineering Course

All the students, except one, expected a course involving cybersecurity training. The one exception reported to never having thought of software security before and therefore had no expectations for the course. However, 80% of the students commented that developing the skill of security-aware programming was empowering because using security-aware programming made the development of secure code an attainable goal.

4.2.5. Responses to FQ 5: Aspect of the course that will be used going forward

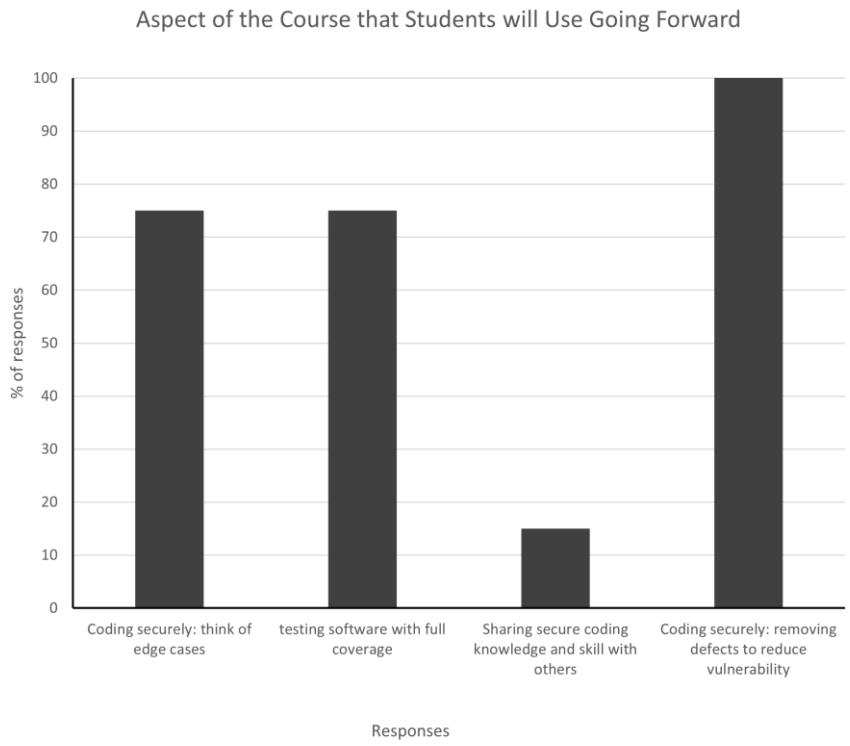

Figure 6. Graph of students' responses to Feedback Question 5

Figure 6 shows students' responses to feedback question 5. All students identified coding securely as a skill that they will continue to use. Some suggested that this would focus on managing code defects while others discussed employing good testing strategies, particularly considering test cases that provide full coverage.

4.2.6. Responses to FQ 6: Impact of taking the Course - 4 Months Later

Only half of the students (10) from the original case study responded to question 6. The 7 seniors had graduated, and 4 other students did not submit a response. Of those who did respond, 100% indicated that they were still using the security-aware programming approach. The strategies being employed the most were robust testing of code to spot errors, input validation, and exception handling. Three of the students reported that they were proudly sharing their knowledge and skill with their peers. One student who is a peer tutor for Computer Science 1, reported that he uses the approach to help underclassmen, during tutoring sessions. However, the greatest endorsement was from the student who wrote "...I cannot help myself now. All my code has to be secure. I cannot code anymore without looking for the security vulnerabilities. I wish that we had been taught this before! Every student should take this course!"

4.3. Other Feedback from Students

In this section, we report on other student feedback from informal face-to-face conversations and emails. Generally, students were excited about learning to code securely. Only three students felt that they had been introduced to the concept of secure coding before. All three of those students

had previously been taught by the researcher. Interestingly, faculty claim that they teach security measures in their courses. However, somehow this had not translated into student perception. Two of the more advanced students said that they had learned some version of secure programming over time, by trial and error. Several students also reported that the content covered in the course had changed how they were programming in their other computer science classes being taken in the same semester. One senior who was in the process of interviewing for a full-time software engineering position said that the security-aware approach to programming was helping with the technical component of their interviews. All the students suggested that some of this content should be included in the introductory courses.

5. DISCUSSION AND CONCLUSION

The case study points to the potential for using TRAC to develop security awareness in students. From student feedback and the observation of student interactions in class, students demonstrated knowledge of software code defects. They also demonstrated the ability to effectively identify and purge those defects from code, to make it more secure. From their feedback, it also appears that students have gained confidence in using the approaches covered in class. They also appear to feel more competent and confident in their ability to write code that is security-aware. All the students who responded to FQ 6 have reported that they are continuing to use the security-aware approach to coding.

One surprise from the case study was that even when faculty think that they are teaching students to code securely, students do not see it that way. This experience emphasizes the importance of designing security content in our courses instead of leaving it to chance. It is also important to present the security content in a contextually relevant way (as is done in the first stage of TRAC). This will help students to understand the rationale for the approach and the trade-offs for not using secure coding measures.

It was also evident that teaching security-aware programming even to seniors, did not always require complex skills like cybersecurity strategies. The basic code defect identification and avoidance appear to be very effective. Often one of the main arguments for not teaching this approach is that faculty are not trained in security. This case study shows that any faculty member who has learned to code can provide opportunities for students to practice and develop the skill for secure programming.

The case study suggests that the TRAC approach can be beneficial to students. However, the size of the study group was a limitation. Also, only half of the tested group responded to FQ 6 - the four-month follow-up. A more complete picture of the impact could be obtained from a larger number of responses.

The current research was designed as a pilot study. In the future, we plan on using this approach to security-aware programming in three other courses: Introduction to Java, Data Structures and Algorithms, and Object-Oriented Programming. This will increase the sample size. We will also gather independent feedback on students' progress, by monitoring their activity using a version control platform. This can be done anonymously, by providing students with random account credentials.

The researchers are aware that teaching students to develop code with security awareness, is not the silver bullet to making all software secure. However, we believe that it is an important tool, that students can use to contribute to the inherent security of their software products.

REFERENCES

[1] Software assurance. [Online]. Available: https://www.cisa.gov/uscert/sites/default/files/publications/infosheet SoftwareAssurance.pdf

[2] B. Martin, M. Brown, and S. M. Christey, "2010 cwe/sans top 25 most dangerous software errors," 2010.

[3] M. Dark, I. B. Ngambeki, M. Bishop, and S. Belcher, "Teach the hands, train the mind ... a secure programming clinic," 2015.

[4] K. L. Nance, B. N. Hay, and M. Bishop, "Secure coding education: Are we making progress?" 2012.

[5] S. Chung, L. Hansel, Y. Bai, E. Moore, C. Taylor, M. E. Crosby, R. S. Heller, and B. Endicott-Popovsky, "What approaches work best for teaching secure coding practices," 2014.

[6] C. Banerjee and S. K. Pandey, "Research on software security awareness: Problems and prospects," SIGSOFT Softw. Eng. Notes, vol. 35, no. 5, p.1–5, Oct. 2010. [Online]. Available: https://doi-org.ezproxy.rollins.edu/10.1145/1838687.1838701

[7] S.-F. Wen and B. Katt, "Learning software security in context: An evaluation in open source software development environment," Proceedings of the 14th International Conference on Availability, Reliability and Security, 2019.

[8] X. Yuan, L. Yang, B. Jones, H. Yu, and B. tseng Chu, "Secure software engineering education: Knowledge area, curriculum and resources," 2016.

[9] A. f. C. M. A. Joint Task Force on Computing Curricula and I. C.Society, Computer Science Curricula 2013: Curriculum Guidelines for Undergraduate Degree Programs in Computer Science. New York, NY, USA: Association for Computing Machinery, 2013.

[10] V. Pournaghshband, "Incorporating the security mindset into introductory programming courses," International journal of advanced computer science, vol. 5, 2015.

[11] K. Frazer, "Building secure software: how to avoid security problems the right way," ACM SIGSOFT Softw. Eng. Notes, vol. 27, pp. 71–72, 2002.

[12] G. McGraw, "Building secure software: better than protecting bad software," IEEE Software, vol. 19, pp. 57–58, 2002.

[13] J. A. Whittaker, "Why secure applications are difficult to write," IEEE Secur. Priv., vol. 1, pp. 81–83, 2003.

[14] B. Taylor, M. Bishop, D. L. Burley, S. Cooper, R. C. Dodge, and R. C. Seacord, "Teaching secure coding: report from summit on education in secure software," in SIGCSE '12, 2012.

[15] M. Bishop, "Learning and experience in computer security education (invited paper)," 2013.

[16] H. Yu, N. Jones, G. Bullock, and X. Yuan, "Teaching secure software engineering: Writing secure code," 2011 7th Central and Eastern European Software Engineering Conference (CEE-SECR), pp. 1–5, 2011.

[17] S. Chung and B. Endicott-Popovsky, "Software reengineering based security teaching," 2010.

[18] B. Taylor and S. Azadegan, "Threading secure coding principles and risk analysis into the undergraduate computer science and information systems curriculum," in InfoSecCD '06, 2006.

[19] M. A. Talib, A. Khelifi, and L. Jololian, "Secure software engineering: A new teaching perspective based on the swebok," Interdisciplinary Journal of Information, Knowledge, and Management, vol. 5, pp. 083–099, 2010.

[20] M. L. Stamat and J. W. Humphries, "Training 6= education: putting secure software engineering back in the classroom," western canadian conference on computing education, 2009.

[21] N. Jones, Q. Yu, K. Schell, and H. Yu, "Teaching secure program design," 2019.

[22] H. Kim, N. Meghanathan, and L. Moore, "Enhancement of an undergraduate software engineering course by infusing security lecture modules," pp. 265–269, 01 2013.

[23] S. Chung and B. Endicott-Popovsky, "Software reengineering based security teaching," in Proceedings of the 7th Annual International Conference on International Conference on Cybernetics and Information Technologies, Systems and Applications (CITSA 2010). Orlando, FL, 2010.

[24] B. Taylor and S. Kaza, "Security injections@towson: Integrating secure coding into introductory computer science courses," ACM Trans. Comput. Educ., vol. 16, pp. 16:1–16:20, 2016.

[25] K. Williams, X. Yuan, H. Yu, and K. S. Bryant, "Teaching secure coding for beginning programmers," Journal of Computing Sciences in Colleges, vol. 29, pp. 91–99, 2014.

[26] E. B. Fern´andez, S. Huang, and M. Larrondo-Petrie, "A set of courses for teaching secure software development," 19th Conference on Software Engineering Education and Training Workshops (CSEETW'06), pp. 23–23, 2006.

[27] B. Taylor, M. Bishop, E. Hawthorne, and K. Nance, “Teaching secure coding: the myths and the realities,” 03 2013, pp. 281–282.
[28] K. Qian, D. C.-T. Lo, R. M. Parizi, F. Wu, E. O. Agu, and B. tseng Chu, “Authentic learning secure software development (ssd) in computing education,” 2018 IEEE Frontiers in Education Conference (FIE), pp. 1–9, 2018.
[29] K. L. Nance, “Teach them when they aren’t looking: Introducing security in cs1,” IEEE Security & Privacy, vol. 7, 2009.
[30] S. A. Ambrose, M. W. Bridges, M. Dipietro, M. C. Lovett, and M. K. Norman, How learning works: Seven research-based principles for smart teaching. Tantor Audio, 2021.

AUTHOR

Rochelle Elva is an Assistant Professor of Computer Science at Rollins College in Florida, USA. Her research interests are Software Quality Assurance, Software Security, Security-Aware Programming, The Personal Software Process, and Computer Science Education.