Security Injections: Modules to Help Students Remember, Understand, and Apply Secure Coding Techniques

Blair Taylor
Towson University
7800 York Rd., Towson, MD, USA, 21252
410-704-4560
btaylor@towson.edu

Siddharth Kaza
Towson University
7800 York Rd., Towson, MD, USA, 21252
410-704-2633
skaza@towson.edu

ABSTRACT
With our global reliance on software, secure and robust programming has never been more important. Yet academic institutions have been slow to add secure coding to the curriculum. We present a model using checklist-based security injection modules to increase student awareness and ability to apply secure coding principles, specifically - identify, understand, and correct key security issues in code. The model is evaluated by mapping assessment questions to the cognitive dimension of the revised Bloom’s taxonomy. Experiments with students in four sections of CS0 and CS1 show that students using our modules perform significantly better at remembering, understanding and applying secure coding concepts. Students exposed to the modules also show increased ability to write code to address specific security issues.

Categories and Subject Descriptors
K3.2 [Computers and Education]: Computer and Information Science Education - computer science education, curriculum, information systems education.

General Terms
Security

Keywords
Security Education, Computer Science Curriculum, Information Security Curriculum Development, Secure Coding, Checklists

1. INTRODUCTION
With the advent of 2011, secure coding is more important than ever. Globally, we are increasingly reliant on software in all fields, including military, medical, financial, and critical infrastructure systems. At the same time, increased connectivity means more malicious attacks. The net effect sets up the scenario for the perfect storm in cybersecurity. The costs are already staggering – the U.S. economy loses one trillion dollars a year to cyber-attacks [8] and the human costs in terms of loss-of-life and health risks are escalating [21].

Security education is a crucial component in addressing the current cybersecurity crisis [4, 7]. However, while many colleges and universities have added security tracks and security courses, the Computer Science (CS) academic community has been slow to incorporate security and secure coding as part of the entire curriculum [12, 16]. The challenges are significant - faculty that are untrained in security, an academic culture that fails to recognize the consequence of software vulnerabilities, a lack of resources, and courses that are overcrowded with difficult topics and struggling students.

For the past four years, we have been developing and implementing checklist-based security injection modules for the introductory courses towards the goal of increasing students’ security awareness and ability to apply security principles. To address the challenges of teaching secure coding we have adapted the following tenets: 1) create a security mindset, 2) start early, and 3) use active learning components.

Our previous publications in this area [5, 14, 15] have focused on student awareness. In this paper we present the results of assessing, not just awareness, but also the ability to apply the concepts learned. In doing so, we map our assessment questions to the revised Bloom’s taxonomy [1] and present results on how well students remembered, understood, applied, and created knowledge related with secure coding in four CS0 and CS1 classes.

2. BACKGROUND
In the past decade, both two and four year institutions have responded to the security crisis by adding security tracks and security electives, which may or may not include secure coding, to the CS curriculum. At our own institution, while the number of students in the security track is increasing, the majority of students do not opt for the track. The problem with electives is that they are a) optional and b) offered late in the curriculum, after students have established programming skills. The end result is that most CS students graduate with inadequate secure coding skills.

2.1 Security in the CS Curriculum
To ensure that all computing graduates are armed with the necessary security skills, including secure coding, the next step is comprehensive security integration throughout the CS curriculum [11,14]. There is evidence of progress in this direction [3] including government initiatives calling for a larger role of education in cybersecurity [20]. The 2008 changes made by the ACM and IEEE to the CS curriculum guidelines include the word "secure" or "security" 136 times [1, 12] which will shape future course development. Based on our research and experience, the strategies for successful security integration include:

Creating a `security mindset`. Many secure coding experts agree that the most critical issue in security education is encouraging a security mindset. The first step is getting faculty and students to recognize the importance of software vulnerabilities and writing secure and robust code.
Starting early. Secure coding education must begin in the first courses required of CS students - Computer Science I (CS1) and Computer Science 2 (CS2) and the prerequisite logic course Computer Science 0 (CS0), if applicable. Teaching secure coding concepts in parallel with programming fundamentals ensures that students establish robust programming skills. One of the fundamental principles of security is to "build security in" and not treat security as an afterthought.

Using active learning. Most college professors, including Computer Science educators, have no formal background in education. As a result, Computer Science and Information Security curriculum is often constructed with little consideration to learning sciences. Niekerk recommends "adherence to sound pedagogical principles when constructing information security campaigns, could improve the efficiency of such campaigns" [10]. In many courses, security concepts are relegated to a sidebar in the textbook. Computer science students have been found to be active learners [9, 17] and we need to ensure that we are using techniques that allow students to assimilate secure coding concepts.

2.2 The Security Injections Project
For the past four years, we have worked with instructors across five institutions to incorporate secure coding concepts into the CS curriculum [5, 13, 14,15,19]. Employing the adage, start early, we have focused primarily on the introductory programming courses required of all CS majors: Computer Science I (CS1) and Computer Science II (CS2), and the preparatory course in programming logic (CS0). Our project goals include:

1) Increase faculty awareness of secure coding concepts
2) Increase students' awareness of secure coding issues
3) Increase students' ability to apply secure coding principles

Towards this end, we have developed and implemented a series of self-contained security injection modules that target key security concepts including integer overflow, buffer overflow, and input validation. Modules are available in different languages, including C++ and Java and are located at http://triton.towson.edu/~cssecinj.

Our project goal of increasing faculty awareness goes hand in hand towards creating a security mindset. We have conducted on-site workshops at each institution to introduce faculty to our modules and discuss the importance of this effort. Currently, we have reached over 60 faculty. Faculty surveys show that instructors felt more comfortable with the concepts after using the modules.

To reach our project goal of increasing students' security awareness and at the same time create a security mindset, each module addresses a key security concept and provides real life examples, to drive home the significance of security vulnerabilities. To ensure that student have fully inculcated these concepts and can apply secure coding principles, we have designed our modules with active learning in mind. The modules were created to ensure that learning progresses at various levels of the Cognitive dimension of the Bloom’s taxonomy with its six categories – remembering, understanding, applying, analyzing, evaluating, and creating [2]. They are modeled on the labs used in the traditional sciences which include background information to create a framework for learning, a hands-on lab activity, and reflective questions.

Another component of our modules and a primary example of the way we employ active learning is the security checklist [13]. Checklists have long been used in aviation for pre-flight checks and more recently have grabbed headlines for their use in healthcare for error reduction and adherence to best practices in clinical care. The security checklist is not “a substitute for analysis” [2], but serves primarily as a learning tool to help reinforce key security principles. Each checklist targets one concept, such as the CS1 buffer overflow example in Figure 1. The self-check process or evaluation required by the checklist requires higher order skills in the learning taxonomy and helps students fully synthesize the targeted skill.

In this paper, we report on results to assess student ability to apply the secure coding concepts. We do this by mapping our

![Figure 1. An example checklist for buffer overflow from the CS1 security injection](image-url)
assessment questions to the Bloom’s taxonomy and using a quasi-experimental design with four sections of CS0 and CS1.

3. ASSESSMENT DESIGN

The assessment was done using two tools - a pre/post survey and a code-check exercise at the end of the semester. The survey contained three kinds of questions:

1. general security awareness: assessed familiarity with common security concerns including phishing, firewalls, and encryption
2. secure coding specific: assessed knowledge on buffer overflow, input validation, and integer error as targeted in the modules
3. demographic and interest in security

The code check exercise included a small code sample (shown in Table 1 followed by three questions (shown later in the section). The intent of the code-check was to test student ability to apply the concepts learned in the modules.

<table>
<thead>
<tr>
<th>Table 1. Code check exercise used in CS0</th>
</tr>
</thead>
</table>
| ```c++
#include <iostream>
using namespace std;

int main()
{
    int roomLength;
    int roomWidth;
    int area;
    int arr[10];
    int i;

cout << "Enter room length and width";
cin >> roomLength;
cin >> roomWidth;

double area = roomLength * roomWidth;
cout << "room area is " << area << endl;
cout << "Enter which element you want?";
cin >> i;
cout << arr[i];

return 0;
}
``` |

We used the guidelines suggested by Thompson et al. [18] to map our assessment tools to the revised Bloom’s taxonomy. Table 2 shows the mapping of the pre and post vulnerability-specific survey questions.

<table>
<thead>
<tr>
<th>Table 2. Classification of the pre/post survey questions within the cognitive process dimension of the Bloom’s taxonomy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Question</td>
</tr>
<tr>
<td>Integer overflow occurs when ...</td>
</tr>
<tr>
<td>Integer overflow is caused by:</td>
</tr>
<tr>
<td>Which of the following should your well-designed program do before processing user input?</td>
</tr>
<tr>
<td>Invalid input can come from the...</td>
</tr>
<tr>
<td>Which programming mistake is one of the major vulnerabilities in today’s applications?</td>
</tr>
</tbody>
</table>

Note: The categories are marked as Remembering (R) and Understanding (U)

Similarly, the questions in the code-check results were also classified into the taxonomy. As Thompson et al. [18] suggest, classifying questions related to coding into one category is not easy or appropriate. The first two questions in the code-check were:

1. Review the above code. Mark with a ‘V’ each line of code that has a potential vulnerability.
2. Next to the ‘V’, list whether it is a possible:  
   a. Buffer overflow  
   b. Integer overflow  
   c. Input validation error

We classified both of these as questions in the ‘Remembering’ and the ‘Applying’ category. The applying category was appropriate as both questions required ‘carrying out or using a procedure in a given situation’ [18]. The procedure in this case was the use of the security checklist to identify the lines in the code where the vulnerability was present. However, to do that, students needed to recall the appropriate checklist for each vulnerability thus classifying them into the remembering category.

The third question was:

3. Next to each ‘V’, write the appropriate code or describe the steps to avoid the vulnerability.

This question was classified into the ‘Creating’ category as it required the students to construct a code segment through the application of known techniques. For instance, our injection modules had guidelines on the procedures to write code to prevent a buffer overflow, however, they did not provide explicit code examples to do the same. Thus, even though the students knew the procedure, they were creating the new code.

The survey and the code check were given to two sections of CS1 and two sections of CS0. Two of the sections were ‘experimental’ sections that explicitly used the three security modules and two were control sections. All sections were taught by the same instructor. No special effort was made to not introduce the security vulnerabilities in the control section and the instructor was encouraged to include her usual content when discussing input validation, integer errors and buffer overflows.

In this study, we only focused on the results of the vulnerability specific questions (type 2) in the survey and the code-check exercise. The following hypotheses were tested:

**H1:** The post-survey scores of the experimental sections will be significantly higher than the post-survey scores of control sections.

**H1a:** The pre-survey scores of both sections will show no significant difference (this will imply that initial student knowledge in both groups was the same).

**H2:** Students in the experimental sections will be better at locating potential security vulnerabilities in a given code segment.

**H3:** Students in the experimental sections will be better at identifying potential security vulnerabilities in a given code segment.

**H4:** Students in experimental sections will be better at fixing security vulnerabilities in a given code segment.
4. RESULTS
A total of 109 students provided valid responses to the survey and 93 students completed the code-check exercise over two semesters.

4.1 Comparison of survey scores
The score of each survey was calculated by counting the number of correct answers (we report on just the secure coding related questions here). We picked the Mann-Whitney non-parametric test to compare the mean rank of the scores in the two groups (experiment and control). This was done for three primary reasons – 1) the n for the groups is different, 2) the Kolmogorov-Smirnov and Shapiro-Wilk test showed that the scores were not normally distributed, and 3) the two groups were independent samples. Figure 2 summarizes the results. As can be seen, for CS0 sections (first two bars in the figure), it was found that the difference between the average post-survey scores of the control (n = 32) and the experimental sections (n = 26) was statistically significant ($p < 0.001$). We also found that there was no statistically significant difference in the pre-survey scores of both sections, thus implying that students started at the same level of knowledge. For CS1, there was no significant difference in the post-survey scores between the sections (n = 15 for control and n = 17 for experiment) at the $p < 0.01$ level, though the difference was significant at the $p < 0.10$ level (which we are considering as mildly significant). However, we also found that in the pre-survey, the control section was significantly higher than the experimental section, implying that the students already knew more about the vulnerabilities than the experimental section at the start of the semester. Thus, even though both sections improved at the end of the semester and the experiment section scored higher, the difference between the two in the post-tests was only significant at $p < 0.10$. Since the n was small in CS1, drawing strong conclusions is difficult.

Overall, these results look promising and H1 was strongly supported in the CS0 sections. Since the survey questions were all in the remembering and understanding category of the revised Bloom’s taxonomy, this implies that the security modules allow students to remember and understand the common vulnerabilities better.

We first came up with an answer sheet that was used by two independent graders to grade the exercise. The scores of each question were tallied and the Cohen’s Kappa statistic was used to compare the agreement in the scores assigned by both graders. During the first round, a low level of agreement was found and we revised the answer sheet to be more specific with respect to the point deduction strategy. For the first question - locating vulnerabilities in code - students were scored from 0-5, specifically 0-2 for identifying potential integer errors, 0-3 for identifying the need for input validation, and 0/1 for identifying potential buffer overflows. For the second question, which entailed mitigating the vulnerabilities, a score of 0-2 was assigned for doing the correct input validation and 0-2 for checking and preventing the buffer overflow (in this question students could write code or describe in plain English how to address it). A second round grading led to an agreement of 89.00% in the grade for locating vulnerabilities (across both semesters), 76.70% for identifying vulnerabilities, and 84.90% for fixing vulnerabilities. As suggested in the literature [6], these are considered substantial levels of agreement. Since the levels of agreement were high, we selected one of the rates randomly for further analysis. The results (shown in Figure 3) are based on combining the two control sections (CS0 and CS1) and experimental sections for clarity.

We found that students in experimental sections (n = 35) performed significantly better ($p < 0.05$) than students in the control sections (n = 48) at locating the potential vulnerabilities in code. This supports H2 and provides further evidence that the modules (and specifically the security checklists) allow students in the experimental sections to remember and better apply the security concepts they learned.

Figure 3. Average scores for control and experimental sections in exercise to locate, identify, and fix vulnerabilities in code

There was also a significant difference in the total scores for identifying the vulnerabilities in code among the two sections (with the experimental section scoring higher). On closer examination, it was found that there was a significant difference in identifying input validation and buffer overflow vulnerabilities (the experimental section scored more) though scores in integer overflow were similar. This may be due to the fact that it is easier to locate integer overflow as its name gives clues on where to find it in code. This supports H3 and points to the advantage of using the modules to introduce the vulnerabilities.

4.2 Comparison of Code-check results
Scoring the code-check presented a different challenge since students used different techniques to fix security vulnerabilities.
As can be seen in the figure, although both sections scored low in fixing errors, the experimental section scored significantly better. Even though this supports H4, the results show that there is significant work to be done in teaching CS0 and CS1 students the techniques to prevent buffer overflows in code. We found that most students in the experimental sections recognized and attempted to mitigate the buffer overflow, however, most attempts were incomplete.

5. CONCLUSIONS
We present a model using checklist-based security injection modules to increase student awareness and ability to apply secure coding principles, specifically - identify, understand, and correct key security issues in code. Experiments with students in four sections of CS0 and CS1 show that students using the modules are significantly better at remembering, understanding and applying secure coding concepts. Students exposed to the modules also show increased ability to write code to address specific security issues.

While previous studies [5,13-15] demonstrated improved security awareness in students across five institutions, the most recent study, which assesses students’ ability to apply secure coding techniques, was limited to one instructor and two split (one control and one experimental) sections in both CS0 and CS1. This assessment is particularly challenging due to the laborious nature of the grading, but further studies are in progress to validate the generalizability of these results.

6. ACKNOWLEDGMENTS
This project is supported by the NSF Course Curriculum and Laboratory Improvement (CLLI) grant number DUE-0817267. We greatly appreciate our colleagues and partners at Towson University, Bowie State University, Harford County Community College, Community College of Baltimore County, and Anne Arundel Community College. In particular, we are grateful to Harry Hochheiser, Mike O’Leary, Shiva Azadegan, AC Chaplin, Patricia Gregory, Jack McLaughlin for their contributions.

7. REFERENCES