Comparing Educational Approaches to Secure Programming : Tool vs. TA

Madiha Tabassum, Stacey Watson, and Heather Richter Lipford
Department of Software and Information Systems
University of North Carolina at Charlotte
9201 University City Blvd
Charlotte, North Carolina
[mtabassu, swatso50, Heather.Lipford]@uncc.edu

ABSTRACT

The cause of many security problems is vulnerabilities in the underlying code. These vulnerabilities are the result of security mistakes made by programmers during application development, often because of lack of knowledge of the security implications of their code. Thus, educators need to teach students as future developers not only how to program, but how to program securely. Many researchers advocate integrating secure programming guidelines across the computer science curriculum. We are exploring a tool to support this goal. ESIDE (Educational Security in the IDE) is an Eclipse plug-in for Java which provides instant security warnings, detailed explanations, and auto-generated remediation code. The goal is to provide contextualized awareness and knowledge of security vulnerabilities and mitigations as students work on programming assignments within any course. In our latest study, we compare our tool against an alternative approach of using security clinics, a one-on-one session with a teaching assistant. We report preliminary findings regarding strengths and weaknesses of our tool based approach to train developers.

1. INTRODUCTION

Software security vulnerabilities are a critical problem that put both an organization and its users at the risk of security attacks and data breaches. Software developers who produce the applications are primarily responsible for preventing such vulnerabilities by employing secure coding techniques. Yet, secure programming knowledge and techniques are not regularly taught in computing curricula. Security education researchers have recognized the importance of integrating security education throughout the computing curriculum to train future developers [1, 2, 3]. We have been investigating one such approach - a tool to provide vulnerability warnings and secure programming education in the IDE while students are working on programming assignments [4, 5]. ESIDE, Educational Security in the IDE, is a Java plugin for Eclipse which is designed to help students address the security implications and teach vulnerability mitigation techniques while they write code. We have performed multiple evaluations of ESIDE in a variety of courses, demonstrating that ESIDE does increase students awareness and knowledge of secure programming. We have also determined the importance of incentivizing learning about secure programming, and carefully timing the introduction of such concepts and skills through our tool [5].

In this paper, we report on work in progress to compare ESIDE against a similar approach utilizing person-to-person feedback on security for student assignments, referred to as a security clinic [6, 7]. Our goal is to understand the strengths and weaknesses of our tool-based approach, and inform design ideas to improve student engagement and interaction with our tool based on similar human-assisted feedback. We describe the lessons learned with this study so far, and the plans for additional data gathering.

2. BACKGROUND

As future software developers, students need to have sufficient knowledge of how to prevent common security vulnerabilities in order to develop secure software. "The ability to write secure code should be as fundamental to a university computer science undergraduate as basic literacy" [8], yet secure programming is often taught too little and too late in a computing program. As such, a number of researchers are investigating various methods for integrating secure coding practices into the computer science curriculum [9], through techniques such as security modules, hand-on-activities, and assignments [1, 2, 3]. However, there will likely be only a limited amount of time within any particular programming course to explicitly cover secure programming. As such, other techniques, including ours, focus on developing these skills alongside course content and programming assignments.

Bishop et al. propose the use of security clinics to reinforce secure programming habits within assignments [6, 7]. Security clinics are similar to writing clinics. Students send their code to the clinic, the clinic members review the code and then have one-to-one discussions with students about the security implication of their code. This approach provides a rich interaction between students and clinic members, and in their studies they found that almost all of the participants who attended the clinic fixed the problems pointed out by the clinician. In other words, the security clinic was found to increase awareness and understanding of the potential problems in students' code [6, 7]. Though this approach

Symposium on Usable Privacy and Security (SOUPS) 2017, July 12–14, 2017, Santa Clara, California.

has been shown to be effective in improving student's secure programming behavior, it can also be time-consuming and resource intensive. Skilled clinicians, such as teaching assistants, need to take the time to first review each student's code and then attend a session with each student. This approach would require universities to have a sufficient number of skilled clinicians to support the program.

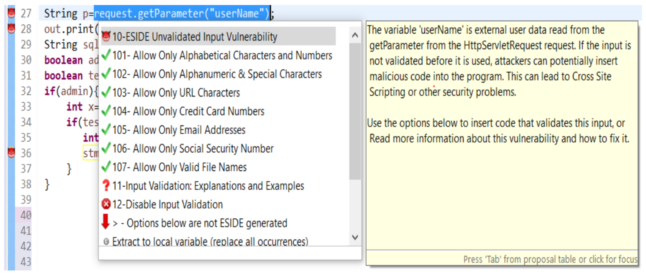
We have proposed a tool-based approach, where students are provided with immediate security-related instructional feedback in their IDE as they develop their assignments. In this way, secure programming knowledge can be introduced early and reinforced throughout a students' education. We provide an overview of the tool below. We have previously evaluated ESIDE in a 3-hour laboratory study[4] and two field studies: one for 14 days and another for an entire semester with students in an advanced web development course[5]. We observed a significant increase in students' secure programming awareness and knowledge with ESIDE. Students also reported an appreciation of the tool. However, we also noticed that interaction with ESIDE was very brief, and that almost no students actually modified their code to mitigate the detected vulnerabilities [5]. Not surprisingly, students were most concerned with completing functionality and did not want to impact that functionality with additional security-oriented code. Thus, we are still investigating how to best engage students with learning about and actually performing secure coding behaviors. We decided to compare our approach with the security clinic which has similar goals, yet with human interaction would presumably be more personalized and engaging to students. Thus, in this paper, we provide early results of comparing students' interaction with ESIDE versus their interactions with a teaching assistant (TA) similar to a security clinic to inform design ideas for enhancing our tool.

3. ESIDE

ESIDE is a proof-of-concept Eclipse plug-in for Java, which integrates secure programming support and education into the IDE. ESIDE works in the background and provides instructional intervention the moment students write insecure code. In this way, ESIDE enhances the students' learning experience as they can directly apply the secure programming lessons they learned in the classroom to their coding practices.

ESIDE works by scanning a selected project for code patterns that match predefined heuristic rules of security vulnerabilities. Though there have been many code related vulnerabilities in Java, currently ESIDE includes only a few of the most common, namely vulnerabilities caused by the lack of input validation, output encoding, and dynamic SQL statements. If left unresolved, these code patterns can lead to real and common security vulnerabilities such as cross-site scripting (XSS) and SQL Injection.

Once ESIDE discovers a potential vulnerability, it places an icon in the left margin of the code editor beside the vulnerable line of code. Hovering over the marker reveals a short message about the type of vulnerability and a single click reveals a list of options for learning more and possibly adding code to resolve the vulnerability. Each item on the list is accompanied by an explanatory pop-up, as shown in Figure 1. For a more in-depth explanation about that vulnerability, students can go to the explanation and example



```
String p=request.getParameter("userName");
out.print(
String sql
boolean ad
boolean te
if(admin){
    int x=
    if(tes
        int
        stm
    }
}
```

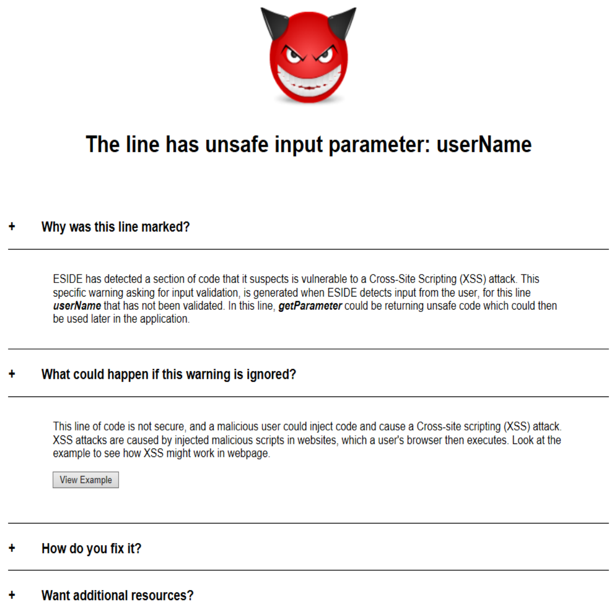
- 10-ESIDE Unvalidated Input Vulnerability
- ✓ 101- Allow Only Alphabetical Characters and Numbers
- ✓ 102- Allow Only Alphanumeric & Special Characters
- ✓ 103- Allow Only URL Characters
- ✓ 104- Allow Only Credit Card Numbers
- ✓ 105- Allow Only Email Addresses
- ✓ 106- Allow Only Social Security Number
- ✓ 107- Allow Only Valid File Names
- 11-Input Validation: Explanations and Examples
- 12-Disable Input Validation
- > - Options below are not ESIDE generated
- Extract to local variable (replace all occurrences)

The variable 'userName' is external user data read from the getParameter from the HttpServletRequest request. If the input is not validated before it is used, attackers can potentially insert malicious code into the program. This can lead to Cross Site Scripting or other security problems.

Use the options below to insert code that validates this input, or Read more information about this vulnerability and how to fix it.

Press 'Tab' from proposal table or click for focus

Figure 1: ESIDE provides a list of suggestions that can be applied to learn about or address the selected warning.



The line has unsafe input parameter: userName

+ Why was this line marked?

ESIDE has detected a section of code that it suspects is vulnerable to a Cross-Site Scripting (XSS) attack. This specific warning asking for input validation, is generated when ESIDE detects input from the user, for this line ***userName*** that has not been validated. In this line, ***getParameter*** could be returning unsafe code which could then be used later in the application.

+ What could happen if this warning is ignored?

This line of code is not secure, and a malicious user could inject code and cause a Cross-site scripting (XSS) attack. XSS attacks are caused by injected malicious scripts in websites, which a user's browser then executes. Look at the example to see how XSS might work in webpage.

View Example

+ How do you fix it?

+ Want additional resources?

Figure 2: Portion of input validation explanation and example page.

page where they can read more detailed information about what the vulnerability is, what could happen if it is left unresolved, and examples of how to fix that vulnerability as shown in Figure 2.

ESIDE also auto-generates remediation code for vulnerabilities caused by lack of input validation and output encoding. Students can choose any of the remediation options as quick fixes from the pop-up list by double clicking on that option. ESIDE will then insert the corresponding input validation and output encoding routine in the code available from ESAPI, an open source secure application development library [10]. For dynamic SQL statements, ESIDE does not provide any quick fixes to generate code, and instead provides a detailed explanation about how to write prepared SQL statements [5]. ESIDE's messages have been iteratively developed over the course of multiple evaluations with students, to communicate clear, simple, and concrete information about vulnerabilities and their mitigations.

4. USER STUDY METHODOLOGY

In this study, we examine the strengths and weaknesses of our tool and explore design improvements that may better engage students by comparing against a similar idea of a security clinic using a teaching assistant. The study goals are to examine the following questions:

- How do students interact with ESIDE versus the TA to learn about security implications of their code?
- How does ESIDE help students advance their secure programming knowledge versus what they learn from the TA?
- What are the questions frequently asked by the students? Can ESIDE provide the answers to those questions?
- How do students incorporate what they learned from ESIDE versus a TA into their assignments?

4.1 Participants and Procedure

For the study, we recruited 23 students from a Network-Based Application Development (NBAD) course that was offered to both graduate and undergraduate students by our university in Fall 2016. In this course, students were assigned to develop a Java-based web application using Java Servlet technology as their final project. While students may have taken an introduction to security course, that course does not cover secure programming. The course instructor assigned students to read a brief Web-based tutorial, developed by us, where they were introduced to basic secure programming techniques (i.e. prepared statements, input validation and output encoding), and received no additional secure programming training in the course. The students were offered either extra credit points or a $10 amazon gift card for participating in the study. From past studies, we learned the importance of having incentives for security as part of the course and assignment. Thus, the instructor agreed to provide such incentives in the assignment requirements. However, we did not realize until we were starting the study that the only assignment requirement was to use prepared SQL statements; there were no additional security-related requirements such as for input validation.

Half of the participants (11 students) were assigned to use ESIDE while the other half (12 students) were scheduled for a one-to-one meeting with a TA, for a one time, 30 minute session. Students were not informed prior to arriving for the study whether their session would be with ESIDE or with the TA. The two TAs were recruited from both current and former TAs and were knowledgeable in secure programming. The students and the TAs signed a consent form prior to participating.

4.2 Study Setup

The ESIDE session took 15-20 minutes of code investigation with ESIDE and 5 minutes of a post-test and survey. Each participant was first given a brief tutorial on how ESIDE works by being shown an example vulnerability and how to fix it in a sample project. Because students did not use the Eclipse IDE for the course (Netbeans is used), we preloaded their assignment code into the Eclipse IDE with the ESIDE plug-in on our study machine prior to the session. The participants were then asked to go through their code, investigate ESIDE's warnings and explanations, and were invited to fix the vulnerabilities in their code. Interactions with the tool were captured by screen recordings and the session was audio recorded. We exported from Eclipse and emailed the projects back to the students if they made any changes and wanted the updated code.

The TA session took 15-20 minutes of discussion on the vulnerabilities in students' code and 5 minutes of a post-test and survey. The TA reviewed the student's code prior to the session for a few minutes to find vulnerabilities. We chose not to have the TA utilize ESIDE for discovering vulnerabilities as we did not want to complicate the session or have the TA overly influenced by our tool. During the session, the TA went over the code with the student, discussed the findings, and answered questions regarding how to fix the vulnerabilities and addressed any other security concerns students had. This conversation was audio recorded, and interactions with the code were captured in screen recordings.

4.3 Post-Test and Survey

Participants were asked to take a paper-based post-test and survey at the end of their session. The post-test questions were the same for both conditions and consisted of three questions asking to identify whether a code segment is vulnerable to SQL injection or required input validation or output encoding and why, as well as three more general questions about SQL injection, input validation, and output encoding. For each question, we asked participants to mark the level of confidence in their answer using a Likert scale. The following is an example of two test questions:

1. What is the purpose of output encoding?
2. Why might the following declared variable require subsequent validation?
   String title= request.getParameter("Title");

In addition to secure coding test questions, we also asked participants for subjective impressions and recommendations about either ESIDE or the TA session.

5. RESULTS

There were 23 Master's student participants: 13 male and 10 female. Eleven participants used ESIDE, 12 saw the TA. We had two TA participants; one was a male undergraduate student who previously served as a TA in our "Software Security Penetration Testing" course and the other was a male Ph.D. student who does research in software security and had previously taught the NBAD course.

Unfortunately, students did not respond to our initial recruiting emails. A follow-up email by the instructor, and the offer of extra credit, did prompt a number of students to volunteer for the study. However, we discovered that they did so only after their projects were completed and about to be turned in. And, the project requirements did not mention security other than the use of prepared SQL statements. Thus, students were not motivated to make changes to their code to fix any security vulnerabilities identified during the session. Without this incentive, the session was merely an awareness building exercise, rather than something driven by the students' needs. This clearly had an impact on how students interacted with ESIDE or the TA, reducing the usefulness of our comparison. Still, we report here on what occurred and what we feel we have learned so far.

5.1 Summary of ESIDE Sessions

We used screen recordings to examine student interactions with ESIDE. Overall, 384 input validation warnings were viewed by the eleven participants, which constitutes an average of 34.91 warnings per person with a standard deviation (SD) of 21.87. Of the 384 warnings, 286 were clicked (74.47%) and 259 were fixed (67.44%). Of the 259 warnings that were fixed, 179 were correct (69.11%). Interestingly, students often selected the first quick fix (87.25%) even if it was not appropriate. One of the participants did not apply any quick fixes and one undid all the quick fixes he applied.

For output encoding, 70 warnings were seen by the participants, which constitutes an average of 6.36 (SD: 3.14) warnings per person. Of the 70 warnings, 33 were clicked (47. 14%) and 11 were fixed (15.71%). However, participants selected the first quick fix option 7 times to encode the output (63.63%). Six participants did not apply any quick fixes; two of them deleted and commented out the vulnerable lines instead of using quick fixes. One of the participants applied quick fixes for one warning and then undid it and disabled all of the markers for output encoding using the "Disable Output Encoding" option. Students did not manually fix any of the warnings. Instead, all the warnings that were fixed were handled by choosing a quick fix which automatically generated the validation or encoding routine.

We also examined student interactions with explanatory pop-ups available directly with the marker and via the "explanation and example page." For input validation, of the 286 warnings that were clicked, students investigated the available information for only 73 warnings (25.52%). Most of the time participants examined the labels of the quick fixes to make the decision about which validation code to insert rather than reading the explanative pop-ups available with each quick fix. Among 11 participants, 9 read the explanation of the title (pop up shown in Figure 1) at least once during the session and of the 2 students who did not, one read similar information in the "explanation and example page". Only 4 of the participants clicked on the "explanation and example" option. One did not read anything and immediately closed the page. The other 3 read the page for an average of 93.33 seconds.

For output encoding, among the 33 warnings that were clicked, participants investigated the information available for 14 warnings (42.42%). Most of the time, students ignored the output encoding warnings because they were printing data only for debugging purposes. Out of 11 participants, 8 read the description of the title and only 1 of the participants went to the "explanation and example page" to review it for about 15 seconds.

In the study, almost all the participants used prepared SQL statements as it was one of the project requirements. Only two students received the SQL Injection warning, one of them three times. This student clicked on the warning each time and went to the explanation page and then reviewed the explanation page for 140 seconds. Another student, who had only one such warning, clicked the warning but did not review any information available in ESIDE. Neither of the students attempted to write prepared statements during the session.

To summarize, students viewed and interacted with a large number of warnings, and tended to go through each file top to bottom looking at all they encountered. Yet participants did very little reading and exploring of the information available. Instead, they were more concerned about fixing all the vulnerabilities in their code as quickly as possible by using quick fix options. They often chose the first quick fix, which was a generic option but not appropriate in all cases. While ESIDE is giving students much awareness about the exact locations of vulnerabilities in their code, we suspect that students are not learning enough to properly fix them. And given the lack of intent on keeping the modified code, were not concerned with whether or not they could actually fix the vulnerabilities found.

5.2 Summary of TA Sessions

We used screen recordings to examine participants' interaction behavior with the TAs. We provided TAs with participant's code 5-10 minutes before the session. During the session, the TAs went over the code with students and discussed the identified vulnerabilities with them. They also explained vulnerabilities to students by providing both explanations of how they can be exploited by attackers in general and how specific vulnerabilities that resided in students' code (on Avg. 2.33 specific lines of vulnerable code per session) affect the security of their applications. The TAs also explained how to prevent those attacks. Students asked TAs for clarification an average of 1.33 times per session when they were confused about the explanation. The following is an example conversation between a TA and a participant:

TA: In place where you are using a variable someone else typed in, not something that is created by the program itself. You should always use COUT or some other sort of encoding thing to prevent XSS.

Participant: Ok.

TA: Something like this. This is generated by one of your functions, you wouldn't ....

Participant: This would go in the COUT thing?

TA:You wouldn't have to put COUT into it because your are generating it by yourself. So, it's safe.

Participant: So, if something generated by something else we need to use COUT?

TA: Yes, user input. Like if you make a account with user name and you wanted to say hello user name. You would wanna use COUT that in case someone have put a script there or something.

One student asked for the exact line of validation code for an input; however, the TA had to search the Internet before he was able to show the student the input validation code. Students also asked questions about security attacks and other security issues an average of 1.67 times per session, particularly about passwords, salting, and hashing. The TAs responded to students' questions by explaining with relevant examples. For example,

Student: One thing which I wanted to know is we type the password into the database. You store it somewhere right... like when a new user want to register at the website, so he needs to give his passwords, we have to encrypt it right... How would you do that?

The TA then proceeded to explain hashing and salting. Despite these two examples of TA-student interaction, much of the session and conversation was driven by the TA.

The TAs discussed different types of security concepts, vulnerabilities and attacks such as Cross-Site Scripting (XSS), SQL Injection, Man-in-the-Middle, Access Control and Denial of Service. However, information provided to the students differed a great deal between the TAs. TA1, the undergraduate who was a TA for a security class, spent most of the session seeking out vulnerable spots in the student's code by looking at the code and by asking participants how the application works (on average 6 times per participant). Only a small portion of time was spent on explaining the vulnerabilities and how they worked. But the time that was spent, was spent on specific vulnerable lines of code and how to fix them. TA1 described SQL injection in only two sessions: One when a participant asked about it and another when he found a missing prepared statement. However, this TA's interactions with students revealed that participants who used prepared statements did not necessarily understand how SQL injection attacks work and how a prepared statement prevents such attacks. To prevent XSS, TA1 always suggested COUT instead of output encoding, and discussed input validation only in two sessions in a single sentence.

In contrast, TA2, a Ph.D. student who had previously taught the course, talked about input validation, output encoding, and SQL injection in detail even if he did not find specific examples of such vulnerabilities in the student's code. In contrast with TA1, he did not spend much time learning about how the code works (on average 1.33 question per participant). Though some students already used COUT, TA2 always suggested output encoding as remediation for XSS. TA2 spoke significantly more, and covered concepts more generally. Thus, he treated his sessions more like general education sessions, while TA1 used his sessions more to find and fix specific issues in the code. This experience emphasized how different a session would be based upon the clinician, and how and what they were trained to provide students. Again though, the effectiveness of the session was greatly reduced by the lack of incentives for students to remember and utilize what they learned.

5.3 Student Perceptions

5.3.1 About ESIDE Session

Almost all the students (n=10) reported that they liked ESIDE suggestions for improving security in their code and appreciated the instant warnings. For example, one student stated:

"It takes and checks all the lines of the code and give users good suggestions to improve their code. That's pretty good."

Three students found the suggestions ESIDE provided easy to understand and helpful. One student stated that:

"The tool was very easy and understood working of the tool very quickly. The representation of bugs also helped in easy identifying."

Two of the students reported that they liked the quick fix options. One stated:

"Double click and the code gets inserted. It shows suggestion on different fixes for a vulnerability. Great tool for a basic version. Saves a lot of time."

Two students stated that they did not like the output encoding warnings that ESIDE generated in their code as they were using the print functions only for debugging. For example,

"I did not like ESIDE giving security breach for system.out. println statement. Because I am using these for debugging the code."

5.3.2 About TA Session

Almost all the participants (n=10) reported that they liked the suggestions given by the TAs about the potential security flaws by reviewing their code. One student stated that,

"Code walkthrough for security issues has performed, and it was very helpful to understand some of the probable code flows that were not fixed."

One felt that the session was an excellent opportunity to ask questions and receive instant feedback. For example,

"Information was clearly delivered. Had the opportunity to ask questions to the tutor and get instant answers."

However, half of the participants (n=6) thought the session was not long enough, and the explanations provided were not detailed enough to clearly understand the idea. One stated:

"Can have the session in detail. Can demonstrate the attack and prevention and doing it in the code given."

We asked students to report how likely they will use the changes in their code made in the session on a Likert scale ranging from 1 to 5 where 1 is very likely, and 5 is completely unlikely. We found that the likelihood for the TA session (on Avg. 1.17) was better than ESIDE session (on Avg. 1.64) and statistically significant at p=0.051 using a t-test.

5.4 Test Results

Participants took a post-test after the session. Given how brief the sessions were, we were not surprised to find no difference in test scores between the ESIDE sessions (48.48% correct) and TA sessions (50% correct). We also analyzed reported confidence in the post-test answers reported in a Likert scale. In general, participants were more confident with correct answers than incorrect answers, yet still rather confident for both correct and incorrect answers. We performed t-test comparisons between the ESIDE and TA test scores and confidence but found no differences.

6. DISCUSSION

From past evaluations, we already knew that at least modest incentives are required to motivate students to learn about and perform secure programming in their assignments [5]. The problems we ran into with the study setup and timing only reiterated this. As such, we intend to fix the incentive and timing and re-run the study as intended and investigate how these different techniques engage students to learn about and practice secure programming. So what have we learned from this study so far? We briefly discuss several takeaways below.

6.1 Vulnerability awareness

The clearest difference between the TA and the tool sessions were the number of specific vulnerabilities covered. ESIDE marked an average of 42 lines of code per participant, and participants visited nearly all of the warnings, giving students insight about the volume of security flaws in their code. Many participants also chose a quick fix, which if chosen properly, would remove the vulnerability from their code. Whereas in the TA sessions, TAs pointed out a little more than 2 specific vulnerable lines of code per participant. This stark difference could be reduced by having TAs use a vulnerability detection tool, allowing them to more quickly find and go through many of the same vulnerabilities. Still, rarely did TAs provide the exact code needed to fix the vulnerability, but instead described the fix more generally.

6.2 Knowledge acquisition

With ESIDE, participants fixed many vulnerabilities by inserting validation code using quick fixes. Though this produced secure code, students spent so little time with the written messages and explanations that they may not really be learning about vulnerabilities and their impact. They may also not be able to fix the vulnerabilities in the absence of the tool. TA explanations were more general, and may provide more detailed knowledge about vulnerabilities. Students were also able to ask questions to confirm their understanding. However, given the lack of incentives of our participants, it is difficult to determine how students would behave if more motivated. We will focus more on this issue when we run the study again. Specifically, we intend to analyze the concepts students are exposed to in each session to examine which concepts may be lacking from our tool.

6.3 Security concerns

In the TA sessions, students were able to ask questions, many of which were not related to secure programming, but instead some other security concept. For example, one student asked,

"Will there be any kind of attack if my website is linking to some external websites? Can something happen over there?"

And another questioned:

"If we input the salt plus the password it gives us the original keyword?"

Over all, students asked 18 questions, and ESIDE could potentially only answer half of those. It would be helpful to obtain data from a bigger population to learn about the common security concerns students have. We could then modify ESIDE to answer more of those questions, and to restructure the information provided by ESIDE to answer questions rather than just provide explanations of menu options.

6.4 Design improvements

As we noticed, students rarely read beyond the label of the quick fixes to choose the correct one. Some labels are more explanatory - such as an email address, or credit card number and students chose these correctly. The more general options, however, such as "Use minimal HTTP characters" which would be appropriate for a password that could have special characters, is not self explanatory. Only one student used this appropriately. We have already changed this option to "Allow only alphanumeric and special characters" as shown in Figure 1. We still need to further improve some of the menu options, in addition to engaging students to understand those options.

Because of the timing of our study, it was not possible for us to determine whether students trust the remediation code they inserted via ESIDE. In the case of input validation, ESIDE added an empty catch block which is executed when user input does not match the validation rule, e.g. insecure inputs. Two students expressed concern that their code might break in the case of exceptions and suggested it would be helpful if ESIDE provides default exception handling. This is an obvious improvement, and we have now added this feature.

ESIDE showed output encoding warnings for Java System.out.println function, which was used only for debugging. To ignore all output encoding warnings, ESIDE provides a menu option "Disable Output Encoding". Instead of utilizing this option, one student deleted and one other commented out all lines that included debugging output. Only one student used the option to disable output encoding warnings. Others just ignored the warnings. We need to provide better ways to help students customize ESIDE output so they do not just start to ignore all warnings when they are not relevant.

7. ADDITIONAL LIMITATIONS

Our study was conducted in a laboratory and has a number of limitations. First, the sample size is still small although on par with other similar studies. Moreover, all of the participants were graduate students from the same course. Students interaction might be different with a larger sample including both novice and advanced students. We also think that students might exhibit more interaction and knowledge about secure programming if the duration of the sessions were longer. We intend to run the future sessions before students are done with their project. This will allow them to actually make use of the knowledge, but may mean that students still need help as they finish their projects. Thus, we are considering in our next study allowing students to return for future sessions as desired and examine what kind of help students seek out when it is provided.

8. CONCLUSION

The risk of security attacks can be reduced by following secure programming guidelines during software development. Developers need to be experienced and knowledgeable enough to apply these guidelines in their work. Hence, educating current and future developers about secure coding guidelines is a necessary yet challenging task. Both ESIDE and security clinics have potential to train students as they learn to program, without adding to their existing course load. Though we believe ESIDE raises awareness about security vulnerabilities, students do not yet engage much with the tool. We plan to extend and continue our study in the coming months to more fully answer our research questions and improve our understanding of how to teach secure programming.

9. ACKNOWLEDGEMENT

Thank you to all participants, TAs and faculty for their time. This work is funded partially by NSF-DGE #1523041 and NSF-CNS #1318854.

10. REFERENCES

[1] Blair Taylor and Shiva Azadegan. Threading secure coding principles and risk analysis into the undergraduate computer science and information systems curriculum. In *Proceedings of the 3rd Annual Conference on Information Security Curriculum Development*, InfoSecCD '06, pages 24–29, New York, NY, USA, 2006. ACM.

[2] K. Nance. Teach them when they aren't looking: Introducing security in cs1. *IEEE Security Privacy*, 7(5):53–55, Sept 2009.

[3] Michael Weeks, Yi Pan, and Yanqing Zhang. Increasing security awareness in undergraduate courses with labware. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*, SIGCSE '16, pages 687–687, New York, NY, USA, 2016. ACM.

[4] Jun Zhu, Heather Richter Lipford, and Bill Chu. Interactive support for secure programming education. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, SIGCSE '13, pages 687–692, New York, NY, USA, 2013. ACM.

[5] Michael Whitney, Heather Lipford-Richter, Bill Chu, and Jun Zhu. Embedding secure coding instruction into the ide: A field study in an advanced cs course. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, SIGCSE '15, pages 60–65, New York, NY, USA, 2015. ACM.

[6] Matt Bishop and BJ Orvis. A clinic to teach good programming practices. In *Proceedings of the 10th Colloquium for Information Systems Security Education*, pages 168–1174, 2006.

[7] M. Bishop. A clinic for "secure" programming. *IEEE Security Privacy*, 8(2):54–56, March 2010.

[8] M. Bishop and D. A. Frincke. Teaching secure programming. *IEEE Security Privacy*, 3(5):54–56, Sept 2005.

[9] S. Azadegan, M. Lavine, M. O'Leary, A. Wijesinha, and M. Zimand. *A Dedicated Undergraduate Track in Computer Security Education*, pages 319–331. Springer US, Boston, MA, June 2003.

[10] OWASP Foundation. Owasp enterprise security api. https://www.owasp.org/index.php/category:owasp_enterprise_security_api, 2016.