Software Reengineering Approach to Teaching Secure Coding Practices

Leo Hansel, Sam Chung, *University of Washington Tacoma,* and Barbara Endicott-Popovsky, *University of Washington*

ABSTRACT

Each year hackers exploit hundreds of vulnerabilities in software, yet the same vulnerabilities continue to appear in code, over and over again, and many educational institutions continue to teach programming as they always have. Companies, such as Microsoft, have found it necessary to conduct secure coding training classes to make up for the absence of the subject in college-level curriculum. Reasons for this lack are many, but our research is motivated by one major barrier: instructor lack of time to convert existing, well-developed curriculum to include secure coding concepts. To address this issue, we have developed an approach that applies the 4+1 Views software-reengineering technique to transform source code that does not incorporate any security concepts, into source code that can defend against attacks. Applying our approach can save instructor time by converting existing assignments into assignments that illustrate secure coding concepts. This approach also helps students understand security problems in code. In our paper, we describe one case study, Cross Site Scripting, as an illustration. We then discuss a workshop conducted for instructors interested in injecting cases such as this into their own courses. Workshop results show that these instructors have strong interest in using our approach to create additional curricula. We plan further refinement and dissemination as well as a comprehensive review of our programming and software engineering curriculum to discover appropriate places for injecting all of the case studies we have developed, as well as supportive topics that lead students to being able to use this technique independently.

Index terms: Secure Coding, 4+1 Views, Re-Documentation Technique, Reverse Software Engineering, Forward Software Engineering

I. INTRODUCTION

Computer defense is not perfect. Anti-virus programs, Intrusion Detection Systems, Intrusion Prevention Systems, and firewalls help increase systems security, but they are only perimeter defenses [1]. Hackers continue to target defects in the design or code of commonly used software [2]. Software developers who lack an understanding of security concepts have a greater chance of producing unsecure source code that can bring disastrous results. The Blaster worm, for instance, infected nearly 330,000 systems worldwide, and was successful because of two-lines of defective code [3].

Defects in software code are not solely caused by a lack of software security knowledge, but can arise from a reluctance to build secure code. Many programmers are concerned more with software product functionality than with software security. Developing secure software is a difficult task and can stress a project's budget. Modern businesses often ignore security practices in favor of a less demanding approach [3].

Any approach to injecting security concepts into programming curricula faces a similar challenge. Many faculty members find it too time consuming to make the needed curricular improvements. Others are unsure about how to incorporate secure coding concepts into existing courses. Still others are simply unaware. Our solution is to educate faculty on how to expeditiously introduce security concepts into existing course curricula artifacts. Adopting a thread approach [2], institutions need only a small budget allocation to upgrade curriculum to include secure coding concepts and faculty members need only to spend a small amount of time to make needed changes.

Our approach uses a specific software-reengineering based technique to transform unsecure source code examples from existing assignments and labs, into source code that can defend against exploits. We develop one case study, Cross Site Scripting (XSS), to demonstrate our approach that employs a re-documentation technique, 4+1 Views [4], to identify attack vectors in the source code, showing where security features can be inserted into legacy code to produce code that is more secure.

II. PREVIOUS WORK

Our approach—introducing reengineering programming assignments using the 4+1 Views—is suitable for teaching because it is less complex than others we have explored and the methodology can detect a broad spectrum of software attacks. Karppinen, et al. [5] applied the Software Architecture Visualization and Evaluation tool (SAVE) to detect security vulnerabilities that violate structural and behavioral system patterns, studying how well static and dynamic analysis can detect such violations and what kind of relationship exists between architecture constructs and security. Byers and Shahmehri developed a modeling

Leo Hansel, MS/CS, Institute of Technology, University of Washington Tacoma.
Sam Chung, Ph.D.,Associate Professor, Institute of Technology, University of Washington Tacoma.
Barbara Endicott-Popovsky, Ph.D., Director for the Center of Information Assurance and Cybersecurity and Research Associate Professor, Information School, University of Washington.

ISBN 1-933510-96-X/$15.00

language, the Security Goal Model (SGM), which can be used in place of attack trees, security activity graphs, vulnerability cause graphs, and security goal indicator trees [6]. Both approaches, while valid and well documented, are examples of techniques that are too complex for most undergraduate computer science students.

Because many approaches used to detect vulnerabilities are deemed too complex for novice programmers to understand, schools offering instruction in secure coding principles, often target upper division students who have completed a number of programming courses [7]. Unfortunately, this occurs after they develop insecure programming habits, making them more difficult to penetrate. We believe that secure coding can be introduced to beginners if coding examples stress the consequences of insecure input and explain the dangers of using insecure function calls [8,9,10].

III. PROPOSED SOLUTION

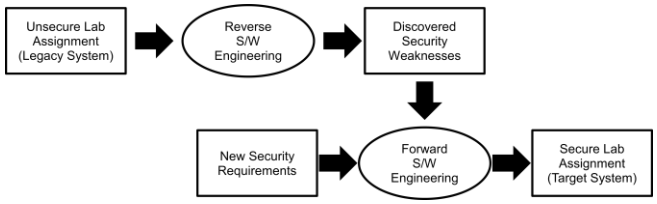

Figure 1. Concept Model of Software Reengineering Approach

We propose that teachers have students use a software reengineering approach to analyze programming assignments, describing source code architecture in the form of 4+1 Views which readily identifies security holes in the code. Once vulnerabilities are found, students modify the 4+1 Views so that they contain protections from attacks. Students then apply secure coding concepts to the source code. By completing these steps, they transform unsecure legacy code into source code that can withstand attacks, Figure 1.

4+1 Views describes the software architecture and consists of the Use Case View (Use Case Diagram), Process View (Sequence Diagram), Logical View (Class Diagram), Physical View (Component Diagram), and Development View (Deployment Diagram). The Use Case View captures the contractual behavior of a system among all stakeholders, enumerating the main success scenario [11]. The Process View represents the behavior of the software system [12]. The sequence diagram shows a set of interacting objects and the sequence of messages exchanged among them [13]. The Logical View presents an object model of the design and shows a set of classes and their logical relationships [3]. The Physical View provides a set of components connected to each other through their interfaces [14]. The Development View shows the static organization of the software in its development environment [3].

Our approach explores attack vectors by focusing on the Use Case and Process Views. Logical, Physical, and Development Views are used as guidelines to understanding the system architecture. The Use Case View presents extensions and exceptions to the main success scenario, which are potential attack vectors [11]. From this view, we can see what developers intended users to do, and what hackers can do, with the system. The Process View contains additional information about the flow of control during the interaction, such as if-then conditions [12] and can show specific locations, within a sequence of method calls, where the system can be exploited.

By creating the complete set of 4+1 Views, students gain a better understanding of the entire software system. Once all views are drawn, students revisit the Use Case View. Instead of acting as a user or developer who engages in normal use of the system, students assume the role of a hacker who aims at breaking the system. Students then create a Misuse Case View that describes ways to exploit the system. Based on exploit paths in the Misuse Case View, students revisit the Process View and identify method calls that render the system vulnerable. By completing these steps, students find ways to take advantage of the system and from there perform desired exploits on executed code. We have introduced security concepts in our own classes using this approach.

Once students understand possible exploits, they revisit the Misuse Case View, adding a security scenario that will fail any attempt to break the system. From there, they revisit the Process View, adding security features (methods) that protect against the exploit. They then inject these security features into the source code, execute the system again, and observe whether the added features prevent the attack. The entire process gives students solid experience in detecting software vulnerabilities and defending systems.

IV. CASE STUDY

We developed three case studies to demonstrate how our software reengineering technique can be injected into course assignments: a buffer overflow attack on an online banking login application, an SQL injection attack on a mailing list subscription web application, and a XSS attack on a discussion board web application. Due to space limitations, we present only the XSS attack.

In a XSS attack, an attacker exploits the trust between a web client (browser) and a server, executing injected script on the browser with the server's privileges [15].The assignment is to create a discussion board web application using J2EE (Java 2 Platform Enterprise Edition). The application displays a textbox for user comments. Once a user clicks "submit," their comments display to other users accessing

ISBN 1-933510-96-X/$15.00 □ 2011 CISSE

the same site. Discussion board comments stay on the website as long as the server is running and will last until the session state expires. Comments display in LIFO (Last In First Out) order: the last comment displays on top of those previous. Figure 2 is a sample discussion board.

In 2006, XSS comprised the largest class of newly reported vulnerabilities [15]. Many web applications are vulnerable to these attacks; some (not all) web application frameworks, such as J2EE and ASP.NET, will interpret script as executable code instead of as pure text, thus our discussion board provides an opportunity for a malicious user to inject JavaScript code into a textbox. This executable code can be a 'Form' tag which can display a login table shown in Figure 3. The full script is shown in Figure 4.

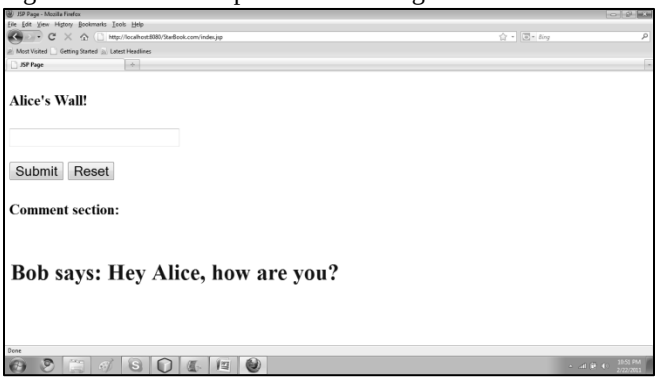

Figure 2. Posting Comments on a Discussion Board

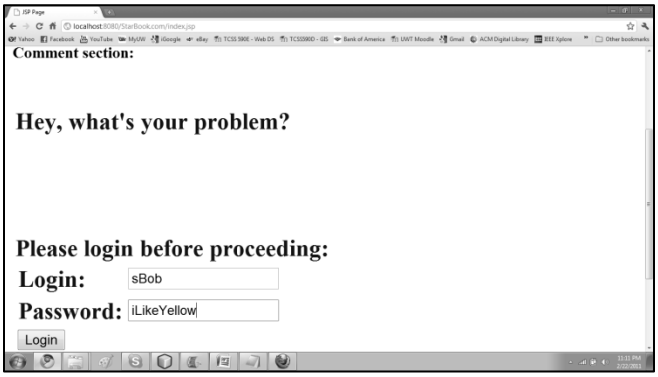

Figure 3. XSS Injection Causes Phishing Login to Appear

Since this is a discussion board, the login form will stay forever unless the administrator of the legitimate website removes it. A user could look at this login form and think that it comes from a legitimate server, tricking the user into entering his/her login credential.

When the victim enters his/her credential and clicks the login button, triggering the 'onClick' event, they will be redirected to a hacker webpage. While redirecting the page, the script also appends a query string with the victim's username/password using the 'document.getElementById' JavaScript function. The hacker can steal the credentials by accessing the query string in the JSP page. To perform this action, the 'request.getParameter' function is used. Once the query parameters are obtained, the hacker stores the credentials in his/her database. The source code to obtain and store credentials in a database is shown in Figure 5.

```
br><br>
Please login before proceeding:
<table>
<tr>
    <td>Login:</td>
    <td>
    <input type="textbox" length="20"
           name="txtLogin" id="idLogin" />
    </td>
</tr>
<tr>
    <td>Password:</td>
    <td>
    <input type="textbox" length="20"
           name="txtPassword" id="idPassword" />
    </td>
</tr>
</table>

<input type="button" value="Login" onClick="varhackerUrl =
'http://69.91.151.164:8080/HackerSite/index.jsp?
loginName=' + document.getElementById('idLogin').value +
'&password=' + document.getElementById('idPassword').value;
varlegitimateUrl = 'http://69.91.151.164:8080/DiscussionBoard/index.jsp';
window.location.href=hackerUrl;
window.location.href=legitimateUrl;
" />
```

Figure 4. Scripting Code of the XSS Attack

```
<%
String stolen_username = (String)request.getParameter("loginName");
String stolen_password = (String)request.getParameter("password");

String url = "jdbc:mysql://******/";
String dbName = *******";
String driver = "com.mysql.jdbc.Driver";
String userName = "******";
String password = "******";
Statement stmt = null;
Connection con;

try {
    Class.forName(driver).newInstance();
    con =DriverManager.getConnection(url+dbName,userName,password);
    String queryString = "INSERT INTO Credentials (username, password)
    values ('" +stolen_username + "', '" + stolen_password + "')";
    stmt = con.createStatement();
    intval = stmt.executeUpdate(queryString);
}
catch(Exception e) {
    System.out.println("e : " + e);
}
 %>
```

Figure 5. Obtaining and Storing Credentials in a Database

Once the stolen credentials are stored, the hacker redirects the page back to the legitimate URL. By doing this, the hacker conceals him/herself from the victim and avoids detection. This attack is similar to a Man-In-the-Middle

(MITM) attack where the hacker places him/herself between the user and the legitimate server, taking over the victim's traffic, gathering the victim's information, and redirecting traffic back to the legitimate server. Our XSS attack lures the victim into entering his or her credentials, steals them by redirecting from a legitimate page to the hacker's page where they are stored.

A. Exploring Vulnerabilities in 4+1 Views

In our programming assignment, students must not only understand how to write source code that meets functional requirements, but also how to secure that software from attack. Once they understand the vulnerabilities of their software code, they create the 4+1 Views, which become a blueprint for transforming unsecure software to secure code. Figure 6 shows the Misuse Case View.

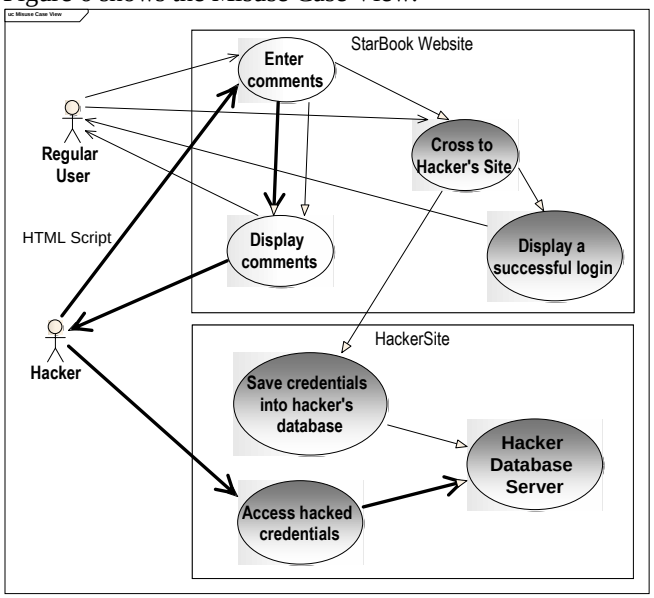

Figure 6. Misuse Case View of the Discussion Board Website

When accessing a website, a regular user intends only to post his/her discussion on the website; however, a hacker has a different intention, as shown in the Misuse Case View, indicated by the thicker lines. A hacker can post a script that makes other users believe that there is a login system they need to go through. From Figure 7, credentials entered by users cross from a legitimate site to the hacker's site where they are captured and stored. Figure 7 shows specifically where, in the interacting objects, the vulnerabilities occur.

The Process View, represented by a sequence diagram, shows interaction between objects in the source code. In the case of a regular user accessing the website, program execution starts when a user accesses the URL of the discussion board. The program calls 'index.jsp' to display the GUI (Graphical User Interface) to the user. When the user submits a comment to be displayed on the website, the program calls a Servlet page to process the request. Once the request is satisfied, the program displays a successful submission to the user and redirects the user back to the legitimate site. In the case of a hacker accessing the website, the flow of execution is similar. The big difference is that the program redirects the user, not to a legitimate site, but to the hacker's site. That flow is depicted in thick lines. The hacker creates the same 'index.jsp' object as the legitimate user. This 'index.jsp' object then calls the hacker Servlet page to obtain credentials and store them in the hacker's database. Once credentials are captured, the program redirects the user back to the legitimate site. These Misuse Case and Process Views identify vulnerabilities.

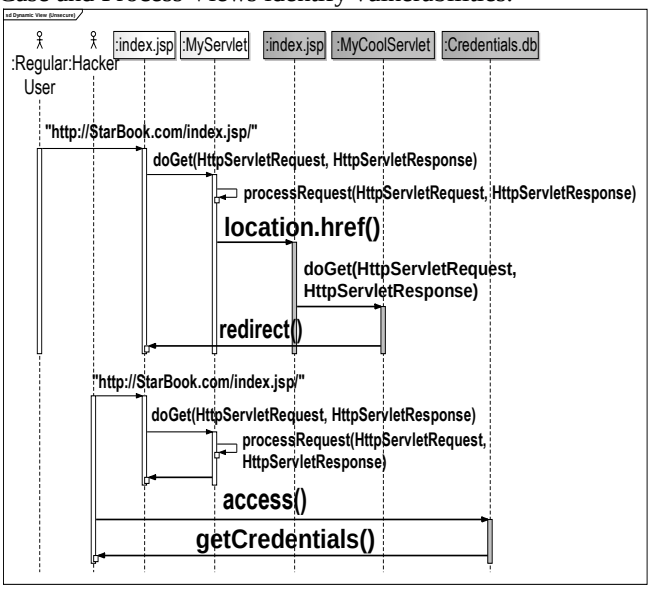

Figure 7. Process View of the Discussion Board Website

In addition to the two views (Figures 6 and 7), students must draw the Logical, Physical, and Development Views. These views do not help in detecting software flaws, but they do serve as a blueprint of the entire software architecture. Figure 8a shows the Logical View, Figure 8b a relationship between classes, and Figure 8c the website class hierarchy.

Figure 8a. Logical View of the Discussion Board Website

The Logical View shows a set of hierarchical classes and their relationships among each other. The highest level of this view consists of just two classes: 'DiscussionBoard' and 'HackerSite.' Students can explore the architecture of each class by digging deep into its inner packages/classes. The relationships between classes are shown by solid or

dashed lines. After the Logical View, students create the Physical View, which is shown in Figure 9.

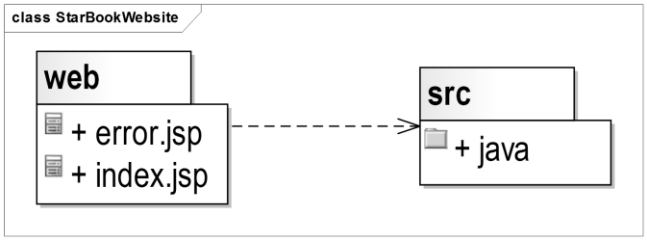

Figure 8b. Relationship between Classes

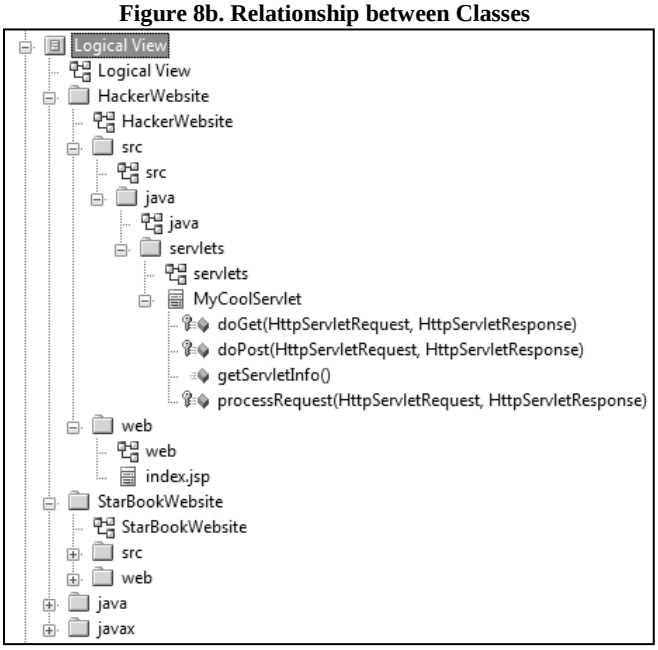

Figure 8c. Class Hierarchy of the Discussion Board Website

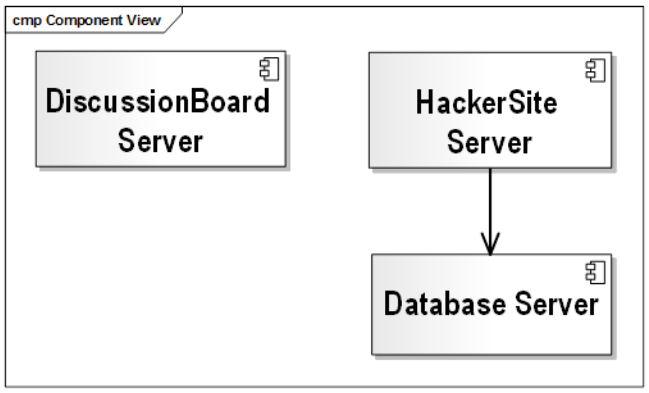

Figure 9. Physical View of the Discussion Board Website

The Physical View shows what components are needed in order to execute the program. In the regular user case, he/she needs only the 'DiscussionBoard' server component; however, in the hacker case, he/she needs both the 'HackerSite,' which references a database server component and the 'DiscussionBoard' server component.

The last view drawn is the Development View, shown in Figure 10, which is represented by a Deployment Diagram showing static organization of the software and the environment in which the software is deployed. From Figure 10, we see that the 'DiscussionBoard.dll' (Dynamic Link Library) file is deployed on the 'DiscussionBoard' server, and the 'HackerSite.dll'is deployed on the 'HackerSite' server. The 'Credentials.db' database schema is deployed on the 'MySQL' database server. The direction arrows indicate the flow of communications. From a client to the server, the communication protocol is HTTP (Hyper Text Transfer Protocol); however, from the server to the database, the communication protocol is ODBC (Open Database Connectivity). By creating a set of 4+1 Views, students discover vulnerabilities in the software that can be exploited. The next step is to transform this unsecure legacy system to a secure target system.

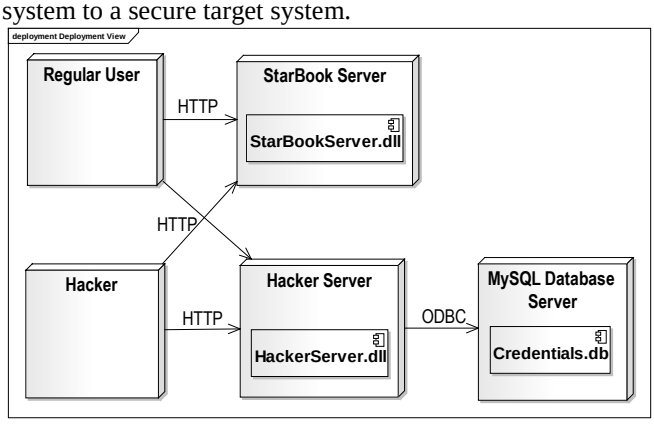

Figure 10. Development View of the Discussion Board Website

B. Transforming Unsecure Software into Secure Code

Before students inject security features into their source code, they draw a Use Case View showing a success scenario that demonstrates secure system behavior among all actors, shown in Figure 11.

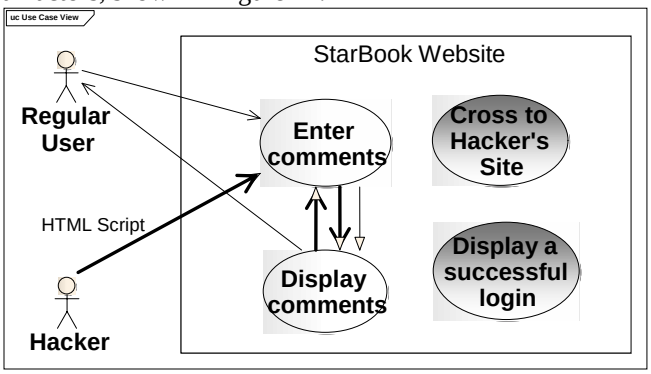

Figure 11. Use Case View of a Secure Discussion Board Site

A secure application executes intended behaviors. Any other behaviors should not happen, or should not be allowed to happen. One method for combating XSS attacks is to apply input validation. In Figure 11, a user can enter comments and receive system notification that these comments are

submitted/displayed successfully. When a hacker attempts to break a secure system, the application performs normally.

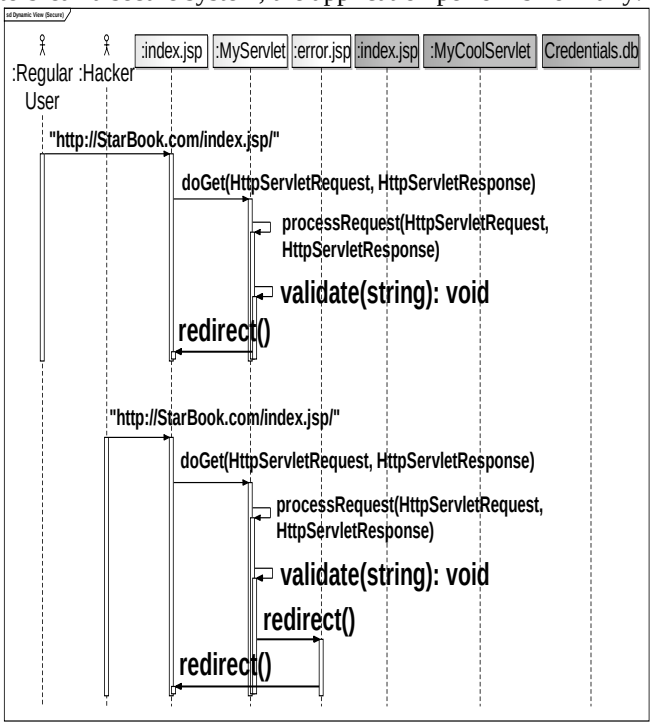

Figure 12. Process View of a Secure Discussion Board Website

Next, students create the secure Process View, shown in Figure 12 above. This sequence diagram will identify specific locations of security holes. In this case, the vulnerability occurs at the 'Servlet' page which processes the request. The unsecure application takes user input and processes it immediately. The secure version validates user input before being passed as a parameter to a method that handles the request. By doing this, the application rejects malicious input and avoids a XSS attack.

The remaining three views of the secure system stay almost the same. The Logical View should reflect the 'validate (string)' method found in the Process View. The Physical and Development Views remain the same.

Once students revisit the 4+1 Views, they must decide how to perform input validation. One method is shown in Figure 13, restricting users from entering malicious characters.

Taking a blacklist approach, the application screens out the following list of dangerous characters [16]: Quotes of all kinds (' and ") – String terminators, Semicolons (;) – Query terminators, Asterisks (*) – Wildcard selectors, Percent signs (%) – Matches for substrings, Underscore (_) – Matches for any character, and Other shell meta-characters (&\|*?~<>^()[]{}$\n\r), which could get passed through a command shell, allowing an attacker to execute arbitrary commands on the machine, Figure 13. (A standard library for input sanitization, like OWASP ESAPI, could be used, as well.)

```
//Returns 0 if the user's input is malicious
Public int validate(String txtField)
{
    StringBuildersb = new StringBuilder();
    sb.append(txtField);
    for(int i = 0; i <sb.length(); i++) {
        if    (sb.charAt(i) == '\" || sb.charAt(i) == '"'   ||
              sb.charAt(i) == ','   || sb.charAt(i) == ';'   ||
              sb.charAt(i) == '*'   || sb.charAt(i) == '%' ||
              sb.charAt(i) == '_'   || sb.charAt(i) == '&' ||
              sb.charAt(i) == '\\' || sb.charAt(i) == '|'  ||
              sb.charAt(i) == '?'   || sb.charAt(i) == '~'  ||
              sb.charAt(i) == '<'   || sb.charAt(i) == '>'  ||
              sb.charAt(i) == '^'   || sb.charAt(i) == '('   ||
              sb.charAt(i) == ')'   || sb.charAt(i) == '['   ||
              sb.charAt(i) == ']'   || sb.charAt(i) == '{'   ||
              sb.charAt(i) == '}'   || sb.charAt(i) == '$'  ||
              sb.charAt(i) == '\n' || sb.charAt(i) == '\r' ) {
                    return 0;
        }
    }
    return 1;
}
```

Figure 13. Preventing Entering of Malicious Characters

When a user enters malicious characters, Figure 14, the application prompts an error page and asks the user to re-enter their information, Figure 15. By validating user input, the application can prevent a XSS attack.

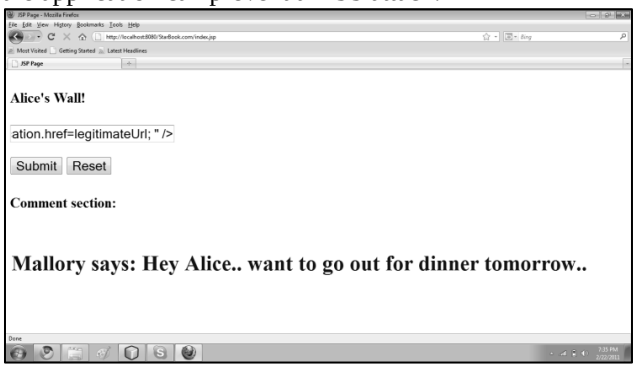

Figure 14. Malicious User Injects JavaScript into a Textbox

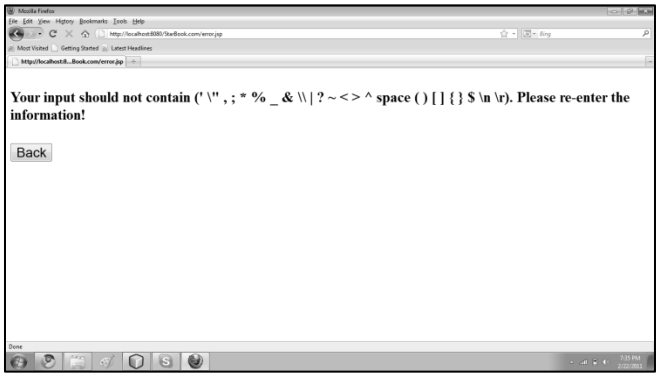

Figure 15. Error Page Indicates Malicious Characters Entered

V. EVALUATION

A two-day secure coding workshop was conducted on August 5 and 6, 2010, at the Center for Information Assurance and Cyber Security at the University of Washington, Seattle, in which these methods and three case studies were instructed: 1) an online banking login application with buffer overflow attack, 2) a mailing list subscription web application with SQL injection attack, and 3) a discussion board web application with XSS attack, discussed in this paper. The workshop targeted computer science instructors and provided information and curriculum artifacts about problems arising from unsecure code and some of the most common or most easily remedied errors, as well as approaches for addressing them. The purpose of the workshop was to introduce instructors to methods and tools that would make teaching secure code concepts and methods easier. By offering explicit reverse engineering instruction using the 4+1 Views approach that we have applied to our own instruction, and by providing specific well-developed examples in the form of actual case studies, we expected to provide instructors a means to overcome lack of time, an apparent major impediment to changing their curriculum to include secure coding practices.

Participating instructors had experience in both industry and academia. Industry experience, reported by eight of the nine participants, ranged from 3 to 30 years, and averaged 14 years. Experience as an instructor, reported by all nine participants, ranged from two quarters to 25 years and also averaged 14 years. Six of the participants described their students as upper division, graduate, or professional workers. Two of these also indicated that they taught only first or second year undergraduate students. Instructors came from both four- and two-year institutions.

Participants were asked to rate how well they understood each workshop component, how relevant the workshop material was to the classes they teach, and how likely they would be to implement what they had just learned in their own courses within the next year. All questions used a scale from 1 (Not at all) to 5 (Completely or extremely). Figures 16 and 17 illustrate their responses to questions asked about their reactions to, and intentions to use, the software reengineering approach and the case studies, respectively. These responses reinforce comments and other data indicating that participants felt least able to incorporate the theoretical and conceptual material presented regarding the software reengineering approach, and most able to apply the case study examples, approaches, and tools, which were seen as more relevant. The major reason for this preference was given as 'lack of time.' The more readily available material—already worked out case study examples—was preferred to the more lengthy process of reverse engineering their own materials.

From the evaluation survey results, Figure 16, the software reengineering approach was above average understood by participants, scoring 3.8 out of 5.0. Also, this approach was seen as somewhat relevant, scoring 3.6 out of 5.0; however, the participants are somewhat less likely to implement the software reengineering approach, applying it to their already existing curriculum materials with a score of 2.8 out of 5.0. Comments indicated that the major impediment to use was their workload.

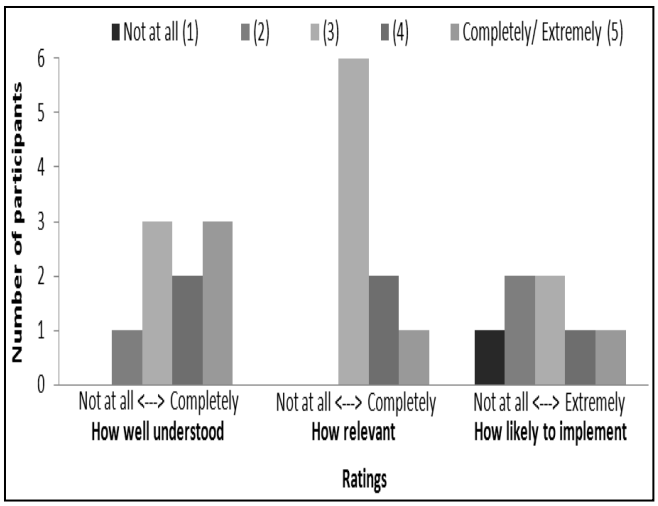

Figure 16. The Reengineering Approach

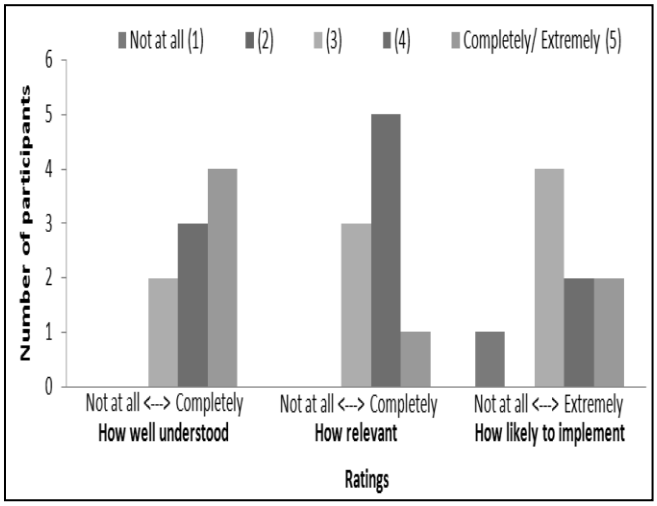

Figure 17. The Cases and Examples

From the evaluation survey results, Figure 17, the case studies were seen as more highly understood, scoring a response of 4.2 out of 5.0, and were regarded as somewhat more relevant to the classes they teach (3.8 out of 5.0). The participants also indicated that they were more likely to incorporate these artifacts directly into their curricula in the next year with a more than favorable response (3.4 out of 5.0). Seven understood the three cases (four "completely"). Six found them relevant (only one indicated "completely" relevant) and the other two selected the scale's midpoint.

Four are likely to implement them – two extremely likely, and four selected the midpoint. One indicated no interest.

VI. DISCUSSION AND CONCLUSIONS

Applying security concepts in the software development process is not just a part of a developer's job; it has to be a discipline that is incorporated naturally in all they do. To efficiently teach security concepts that can reside in students' minds permanently, instructors need preparation and tools so that they themselves can teach these concepts. It appears that many view lack of time as a major impediment to incorporating these concepts into their curriculum.

In our workshop we provided both theoretical and conceptual material and actual cases. While overall, participants appeared satisfied with the workshop, we concluded that the most effective way to prepare instructors is to have many useful cases developed that demonstrate the software reengineering technique—Software Architecture Reconstruction using the 4+1 Views—rather than expect instructors to build their own examples using our approach. In our work, we have found that 4+1 Views is a successful method for providing students a blueprint to understand the structure of software from multiple and parallel views, allowing efficient identification of vulnerabilities and insight into where to insert security features. In the case study presented in this paper, we have demonstrated how to use this approach to transform unsecure legacy software subject to a XSS attack into a secure target system by introducing input validation.

VII. FUTURE WORK

In recognition of instructors' expressed concern for lack of time to alter existing curriculum, and to broaden implementation of this work, we plan a second workshop in which we will continue to teach our reengineering approach, but will expand the numbers of case studies we present in order to provide specific curriculum artifacts for immediate use in attendees' courses. We expect increased interest in implementation of curriculum changes as a result. We also are developing a map of computer science curriculum, indicating where we recommend inserting our approach.

ACKNOWLEDGEMENT

The work discussed in this paper is funded by NSF/SFS Capacity Building grant: DUE-0912109.

REFERENCES

[1] D.P. Biros. (2004). Scenario-based training for deception detection. *Proceedings of the 1st Annual Conference on Information Security Curriculum Development (InfoSecCD '04)*, October 2004, Kennesaw, GA.

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

[3] M. Howard, D. LeBlanc, and J. Viega. *19 Deadly Sins of Software Security*. McGraw-Hill Osborne Media, 2005.

[4] P. B. Kruchten. The 4+1 View Model of Architecture. *IEEE Software*, 12 (6), 1995, pp. 42 – 50.

[5] K. Karppinen, M. Lindvall, and L. Yonkwa. (2008). Detecting Security Vulnerabilities with Software Architecture Analysis Tools. *IEEE Int'l Conference on Software Testing Verification & Validation Workshop* (ICSTW 08), pp.262-68.

[6] D. Byers and N. Shahmehri. (2010). Unified modeling of attacks, vulnerabilities and security activities. *Proceedings of the 2010 ICSE Workshop on Software Engineering for Secure Systems (SESS '10)*, May 2 2010, Cape Town, South Africa

[7] Taylor, B. and S. Azadegan. (2006). Threading Secure Coding Principles and Risk Analysis into the Undergraduate Computer Science and Information Systems Curriculum. *Proceedings of the 3rd Annual Information Security Curriculum Development*, Kennesaw, GA.

[8] Taylor, B. and S. Azadegan. (2008). Moving Beyond Security Tracks: Integrating Security in CS0 And CS1. *Proceedings of the 38th SIGCSE Technical Symposium on Computer Science Education*, Portland, OR, p. 320-324.

[9] Endicott-Popovsky, B.E., Frincke, D., V.M. Popovsky. (2005). Secure Code: The Capstone Class in an IA Track. *Proceedings of the 9th Colloquium for Information Systems Security Education*, Georgia Institute of Technology: Atlanta, GA, pp.100-108.

[10] Bishop, M. and B. J. Orvis. (2006). A Clinic to Teach Good Programming Practices. *Proceedings of the 10th Colloquium for Information Systems Security Education*, University of Maryland: Adelphi, MD, pp. 168-174.

[11] H. Eichelberger. Automatic Layout of UML (2008). Use Case Diagrams. *Proceedings of the 4th ACM Symposium on Software Visualization (SoftVis '08)*, September 16–17, 2008, Herrsching am Ammersee, Germany.

[12] UML 2.0 Infrastructure Specfication. Object Management Group. www.omg.org. Last accessed Jan. 3, 2010.

[13] A. Rounte, O. Volgin, and M. Reddoch.(2005). Static Control-Flow Analysis for Reverse Engineering of UML Sequence Diagram. *Proceedings of the 6th ACM SIGPLAN-SIGSOFT workshop on Program Analysis for Software Tools and Engineering (PASTE '05)*, Lisbon, Portugal.

[14] S. Emadi and F. Shams.(2008). From UML Component Diagram to an Executable Model Based on Petri Nets. *International Symposium on Information Technology (ITSim '08)*, Kuala Lumpur, Malaysia, 2008, pp. 1-8.

[15] G. Wassermann and Z. Su.(2008). Static Detection of Cross-Site Scripting Vulnerabilities. *Proceedings of the 30th International Conference on Software Engineering (ICESE '08), May 10–18, 2008*, Leipzig, Germany.

[16] Ed Skoudis and Tom Liston. *Counter Hack Reloaded: A Step-by-Step Guide to Computer Attacks and Effective Defenses*, 2ND edition, Prentice Hall, 2006.