Journal of Cybersecurity Education, Research and Practice

Volume 2021 | Number 1 Article 5

July 2021

Secure Coding in Five Steps

Mini Zeng
Jacksonville University, mzeng@ju.edu

Feng Zhu
University of Alabama in Huntsville, fz0001@uah.edu

Recommended Citation

Zeng, Mini and Zhu, Feng (2021) "Secure Coding in Five Steps," *Journal of Cybersecurity Education, Research and Practice*: Vol. 2021: No. 1, Article 5.
DOI: https://doi.org/10.62915/2472-2707.1076
Available at: https://digitalcommons.kennesaw.edu/jcerp/vol2021/iss1/5

Secure Coding in Five Steps

Abstract

Software vulnerabilities have become a severe cybersecurity issue. There are numerous resources of industry best practices available, but it is still challenging to effectively teach secure coding practices. The resources are not designed for classroom usage because the amount of information is overwhelming for students. There are efforts in academia to introduce secure coding components into computer science curriculum, but a big gap between industry best practices and workforce skills still exists. Unlike many existing efforts, we focus on both the big picture of secure coding and hands-on projects. To achieve these two goals, we present five learning steps that we have been revising over the last four years. Our evaluation shows that the approach reduces complexity and encourages students to use secure coding practice in their future projects.

Keywords

SECURE CODING, CYBERSECURITY, SECURE SOFTWARE DEVELOPMENT

INTRODUCTION

Software vulnerabilities pose a severe cybersecurity challenge. According to the National Vulnerability Database (NVD), the number of new software vulnerabilities dramatically increased to more than 16,000 every year (CVSS, 2020). Among the vulnerabilities, over 25% of them are of high severity. The exploitation of the vulnerabilities cost $60 Billion every year in the U.S. alone. Companies and organizations have created numerous industry best practices resources, code review methods (Conklin et al., 2017; Leblanc et al., 2003; Rothke, 2006; Taylor et al., 2011), testing guides (Meucci et al., 2013), secure coding standards (Long et al., 2011; Seacord, 2005, 2008), vulnerability databases (*CWE Common Weakness Enumeration*, 2014; MITRE, 2020b), dictionaries of attacks (MITRE, 2020a), the framework for prioritizing weaknesses (Coley, 2014; National Institute of Standards & Technology, 2019) and software tools (Microsoft, 2016; *OWASP ZAP*, 2020; Shostack, 2014; Veracode, 2020b). However, these resources are not designed for classroom usage. When first introduced students to these materials, they found an overwhelming amount of information.

There are academia's efforts to introduce secure coding components into the computer science curriculum (Software Engineering Institute (SEI) at Carnegie Mellon University, 2021; Towson University, 2020; Whitney et al., 2018). Secure software development courses are now offered in several universities, including ours. Organizations and universities made their teaching material available online (Software Engineering Institute (SEI) at Carnegie Mellon University, 2020; Wenliang Du, 2020). For example, Yuan et al. developed secure coding learning modules that focus on manual code review and static analysis on C/C++ and Java code(Dukes et al., 2013; Xiaohong Yuan, 2019). At CMU, SEI provides lecture materials and artifacts (Software Engineering Institute (SEI) at Carnegie Mellon University, 2020). The Security Injection Project at Towson University developed security injection modules integrated with CS0, CS1, CS2, and other courses (Kaza et al., 2010; Towson University, 2020). The SEED lab also provides software security labs online (Du et al., 2007). Instead of focusing on a specific component, we emphasize the big picture of secure coding and provide sample projects to practice the main components. The long-term goal is to educate students on the right mindset, necessary knowledge, and skills to develop secure software.

Our first step started with introducing the big picture of secure coding to students based on the Microsoft Security Development Lifecycle (SDL) (Microsoft, 2012), including seven phases, training, requirement, design, implementation, verification, release, and response. The approach proposed in this paper focuses on five learning steps: 1) gain knowledge of common vulnerabilities, 2) identify vulnerabilities, 3) prioritize vulnerabilities, 4) mitigate coding errors, and 5) document decisions and fixes. This approach guides students to take small steps and go through the process. This approach's specific objectives include introducing industry best practices and hands-on practices of locating resources, manual code review, static analysis tool, and prioritizing vulnerabilities. We also evaluate whether this approach reduces complexity and encourages students to use secure coding practice in their future projects.

The proposed approach has four main contributions. First, students learn a broad set of secure coding skills. Second, students gain knowledge of secure coding resources, including guides, books, vulnerability databases, mitigation methods, detection, validation approaches, and software tools. Third, these steps are easy to follow. Last, the hands-on case studies and videos facilitate other institutes to adopt, especially the manual code review and the free static analysis tool.

The rest of the paper is organized as follows. Section 2 discusses the background and related work. Then, Section 3 describes the five learning steps. Section 4 illustrates the evaluation and students' feedbacks. Section 5 concludes the contributions and presents future works.

BACKGROUND AND RELATED WORK

This section covers background information about secure software development, secure coding practices, and academic efforts to teach secure coding. We discuss secure coding resources (CWE, OWASP, and SAFEcode) and tools that developers use to detect coding errors. Also, we discuss the web application which is used for hands-on practices.

Secure Software Development

Microsoft published a Security Development Lifecycle (SDL), which includes seven phases: training, requirement, design, implementation, verification, release, and response in 2012 (Microsoft, 2012). Recently, the seven phases were revised into twelve practice areas (Microsoft, 2020a). The twelve practice areas are 1) provide training, 2) define security requirements, 3) define metrics and compliance reporting, 4) perform threat modeling, 5) establish design requirements, 6) define and use cryptography standards, 7) manage the security risk of using third-party components, 8) use approved tools, 9) perform static analysis security testing

(SAST), 10) perform dynamic analysis security testing (DAST), 11) perform penetration testing, and 12) establish a standard incident response process. Microsoft also suggests that organizations should adapt rather than adopt the SDL process.

Other than Microsoft Security Development Lifecycle, the National Institute of Standard and Technology published a Secure Software Development Framework (SSDF) (Dodson et al., 2019). The SSDF covered industry practices related to secure coding and other secure software development phases (e.g., security requirement and configuration). SSDF promotes critical secure coding practices such as creating source code adhering to secure coding practices, assessment, prioritization, and vulnerability remediation.

Secure Coding Education

Colleges and universities designated their undergraduate and graduate programs and courses related to software security. Hands-on labs are also designed to integrate into software security-related courses (Xie et al., 2015). The computer science department at Purdue University, for example, offers a "Software Security" course. The course focused on software security fundamentals, secure coding guidelines and principles, and advanced software security concepts. Students learn to assess and understand threats, design and implement secure software systems, and mitigate common security pitfalls (Purdue University, 2018). Yuan at North Carolina A&T State University developed a "Secure Software Engineering" course. The course discusses how to incorporate security throughout the software development lifecycle (Yuan et al., 2012). Her course, "Software Security Testing," focused on software security testing techniques and tools (Yuan et al., 2012). The Laboratory of Information Integration Security and Privacy at the University of North Carolina at Charlotte offered a course named "Software Vulnerability Assessment" (Chu et al., 2009). The course emphasized vulnerabilities and mitigations through secure software design and implementation. Walden and Frank in Northern Kentucky University offered a seminar course - "Secure Software Engineering." The course included a set of secure software engineering teaching modules such as software security, threats and vulnerabilities, and risk management (Walden et al., 2006).

Lecture materials and teaching modules are also developed and shared. The Software Engineering Institute (SEI) at CMU provides lecture materials and artifacts online that faculty can utilize to integrate into their curricula (Software Engineering Institute (SEI) at Carnegie Mellon University, 2020). The SWEET (Secure Web Development Teaching) project developed portable teaching modules for secure web development (Chen et al., 2010). The SEED project included lab exercises for computer security education (Wenliang Du, 2020). The labs include the demonstration of common vulnerabilities, attacks, and applications of security principles and techniques. The Security Injection Project at Towson University developed security injection modules integrated into existing computer science programming courses (Towson University, 2020). CLARK, which Towson University developed, hosts a diverse collection of cybersecurity learning objects and repositories (Towson University, n.d.), including ours.

Educators may reference guidelines for their software security curriculum, courses, or seminars. National Initiative for Cybersecurity Education (NICE) published a Cybersecurity Workforce Framework, which describes the specific knowledge, skills, and abilities. It is required for the work roles related to cybersecurity (National Initiative for Cybersecurity Careers and Studies, 2020). National Center of Academic Excellence Cyberdefense education program published knowledge units to guide cybersecurity educators. It includes a Secure Programming Practices Knowledge Unit and a Software Security Analysis Knowledge Unit with guidance on learning outcomes and topics (NIETP, 2020).

Secure Coding Best Practices

The most effective way is to follow the industry best practices. OWASP offers multiple solutions. The OWASP Software Assurance Maturity Model Project specifies a framework for designing and implementing secure software (Arciniegas et al., 2019). The OWASP Development Guide provides practical instructions and J2EE, ASP. NET and PHP code samples (*OWASP Development Guide*, 2005). OWASP Secure Coding Practices Quick Reference Guide provides a checklist to help developers decrease the vulnerabilities before the software package has been completed (The Owasp Foundation, 2010).

Software Assurance Forum for Excellence in Code (SAFEcode) publishes secure development practices emphasizing real-world actions (SAFECode, 2018). SAFEcode best practices provide more robust controls and integrity for commercial applications during the design, programming, and testing phases. SAFECode includes methods and tools to verify each practice, mitigation, and CWE references for each practice listed. SAFEcode and Cloud Security Alliance released a guide to help readers better understand and implement best practices for secure cloud applications' development (Sullivan et al., 2013).

CERT publishes C, C++, Java coding standards (Long et al., 2011; Seacord, 2014; Software Engineering Institute (SEI) at Carnegie Mellon University, 2016). Companies such as Cisco, Oracle, and Microsoft widely adopt secure coding standards and suggestions (Cisco, 2016; Long et al., 2011; Microsoft, 2020b). In this paper, we introduce students to secure coding standards and teach them how to apply them when developing software.

Vulnerabilities Databases

We introduce multiple vulnerability repositories to the students: Common Vulnerabilities and Exposures (CVE), U.S. National Vulnerability Database (NVD), and Common Weakness Enumeration (CWE). CWE includes a list of software Weakness types that can occur in various stages of software development. The CWE system provides a standard measuring technique for software security tools and a common baseline for weakness identification and mitigation techniques(*CWE List*, 2020). The latest CWE software vulnerability list, CWE list Version 4.0, includes a thousand errors and error categories (*CWE List*, 2020). The CVE system is a categorization of software weaknesses (MITRE, 2020b). Both CWE and CVE are included in the U.S. National Vulnerability Database (NVD). It provides a data repository of known vulnerabilities that can be used for vulnerability management and security compliance requirements.

In 2020, CWE update the 2020 CWE/SANS Top 25 Most Dangerous Software Errors. It lists the most severe and common software errors(CWE Top 25 Most Dangerous Software Weaknesses, 2020). These errors are based on more than 800 programming errors, design errors, and architecture errors, leading to various vulnerabilities. The 2020 CWE Top 25 is formed based on real-world vulnerabilities found in the NVD. According to NVD Count and the average CVSS score, the highest score is given to Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting'). In 2020, there are 3788 entries related to this kind of vulnerability in the NVD data set. The average CVSS score is 5.80. The overall score calculated by the CWE scoring formula is 46.82(CWE Top 25 Most Dangerous Software Weaknesses, 2020). Once attackers use this vulnerability to inject malicious scripts, they could transfer private information, such as cookies that may include session information, from the victim's machine to the attacker (CWE, 2020).

Static and Dynamic Analysis

Static and dynamic analysis are the most popular types of security test tools. Static analysis tools discover security errors without running the program, while dynamic analysis tools examine software by executing the program.

Static analysis tools are much more scalable than manual code review. They can scan a large amount of code and can also be used repeatedly. They automatically find errors such as buffer overflows and SQL Injection and provide mitigation suggestions. Some of the static analysis tools support multiple languages. Agnitio provides static analyses for ASP.NET, C#, Java, Javascript, Perl, PHP, Python, etc. (*Agnitio - Static analysis*, 2015).

Some tools are programming languages specific. For example, OWASP LAPSE+ Static Code Analysis Tool is designed for Java (*OWASP LAPSE+ Static Code Analysis Tool for Java*, 2017; Pérez et al., 2011), FlawFinder for C/C++ (Wheeler, 2017), Pylint for Python (*Pylint - python code analysis tool*, 2020) and RIPS for PHP (RIPS - A static source code analyzer for vulnerabilities in PHP scripts, 2017). Some static analysis tools could be integrated into IDEs. For example, .NET analyzers could be installed in Visual Studio using the Nuget package (Microsoft, 2018). In academia, James Walden and Maureen Doyle developed an indicator named SAVI (Static-Analysis Vulnerability indicator) that combines several static-analysis metrics and ranks web applications' vulnerability (Walden et al., 2012).

We educated students on the static analysis tools and the dynamic vulnerability scanning tools critical for overall program security. The systematic and random approaches often catch the security errors missed by manual analysis and testing approaches. Dynamic analysis tools such as Abbey Scan, WebInspect, HCL AppScan, and Adobe Ride provide security solutions targeting different development life stages (OWASP, 2020). Veracode provides both static code analysis and dynamic web application analysis (Veracode, 2020b, 2020a). Similar to the static analysis tools, these dynamic analysis tools may not be perfect. There are many false-positive cases and may have false-negative problems.

Manual Code Review

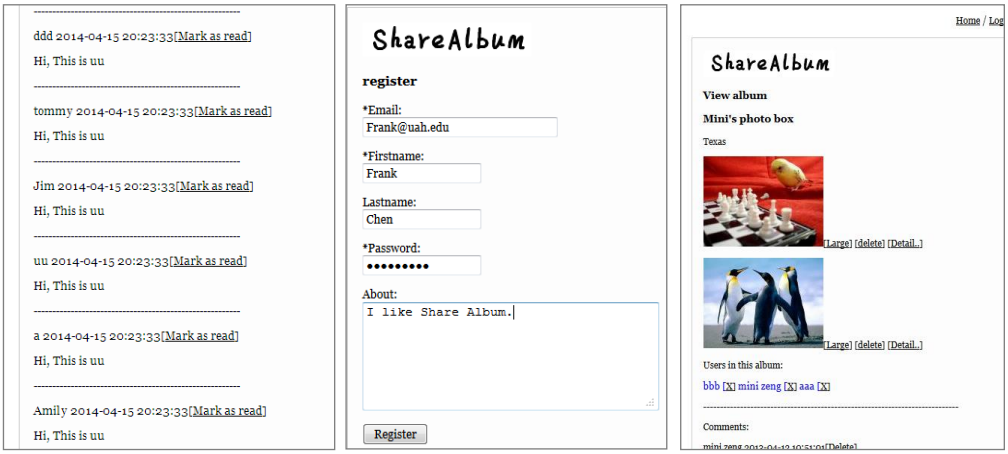
Automated tools/scanners can help to find flaws. However, they cannot discover all vulnerabilities, and often they report many false-positive cases. Hence, manual code reviews are essential. Industry best practices indicate no substitution for manual code reviews because developers understand the environment, context, and users best. Industry and organizations publish guidelines and standards to support manual code review. For instance, the OWASP Code Review Guide focuses on manual code review (Conklin et al., 2017). It suggests a code review checklist covering most critical security controls and vulnerability areas such as data validation, authentication, session management, etc. SEI CERT's coding standards support the development of coding standards for commonly used programming languages such as C, C++, Java, and Perl, and the Android platform (Long et al., 2011; Seacord, 2008, 2014; Software Engineering Institute (SEI) at Carnegie

Mellon University, 2016). Books such as Writing Secure Code (Leblanc et al., 2003) and 24 Deadly Sins of Software Security (LeBlanc et al., 2010) provide best practices on critical items to be review.

Case Study - ShareAlbum

To present the approach in a realistic setting, we provided students a simple and fully functional application named ShareAlbum. It was developed by students who won multiple coding awards (America's Datafest, 2013). The project is available on our website and the CLARK website[1]. The reason we choose ShareAlbum was that the code is simple and thus minimizes the learning curve. We often update the source code to keep up with the new software versions.

ShareAlbum is used to share albums, photos, and videos among users. This application developed using PHP, HTML, and MySQL. The ShareAlbum database stores and keeps track of images, videos, photo-tags, and users' information. In ShareAlbum, the photos and videos could be uploaded and tagged. The albums and videos are categorized as private or public when they are created. Users set privileges to review, make comments, and tag on public photos and videos. Users could send messages to each other, be notified of new messages (Figure 1a). Figure 1b and Figure 1c illustrate the registration page and album view page of ShareAlbum. In the lecture, we demonstrated the components of ShareAlbum to students. A document explaining the design and coding details of ShareAlbum was also shared with students.



(a) (b) (c)

Figure 1 ShareAlbum Received Messages.

[1] URL will be added after the blind peer review.

METHODOLOGY

The proposed approach aims to teach students the big picture of secure coding and offer them hands-on opportunities to apply secure coding best practices when developing software. The five steps of secure coding were taught in a computer and software security course. The course was offered for both undergraduate and graduate students. To offer students a big picture of secure coding, the Microsoft Security Development Lifecycle (Microsoft SDL) phases were taught in the first section of the semester before the five steps of secure coding were applied. The five steps of secure coding practices aim to let students practice secure coding phases, not just knowing them on a conceptual level. The five steps and assessments were completed as five milestones. Hands-on projects were assigned as homework. Tutorials, project description, case study source code, video tutorials, demos (videos), all related materials are accessible online (Zeng et al., 2020).

The learning steps adapt from Microsoft SDL phases. The secure software development framework and the Microsoft SDL practices are integrated into the steps. The proposed five learning steps are: 1) gain knowledge of common vulnerabilities, 2) identify vulnerabilities, 3) prioritize vulnerabilities, 4) mitigate coding errors, and 5) document decisions and errors. Figure 2 illustrates the details of the five learning steps.

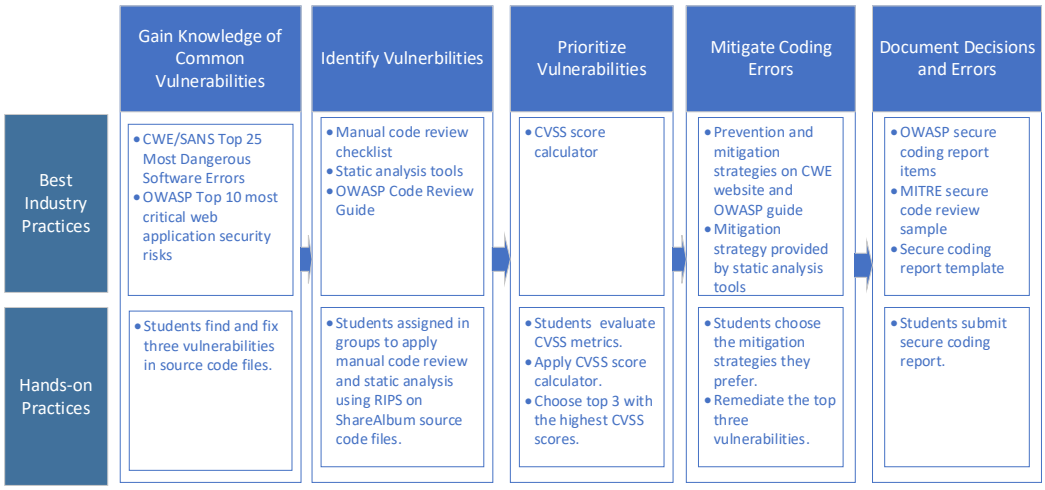


Figure 2 Five secure coding learning steps.

Step 1: Gain Knowledge of Common Vulnerabilities

This step teaches students the most common vulnerabilities. CWE's top 25 most dangerous software errors and OWASP's top 10 most critical web application security risks were introduced in this step. We chose these two lists because they include the current and most widespread and critical errors.

In the lecture, we chose three common vulnerabilities from the lists. The descriptions of each vulnerability, the consequences of each vulnerability, detection method, attack mechanisms, and mitigations were explained at a high level. Then, we demonstrated and explained the vulnerable code, the attack actions, consequences, and detailed mitigation suggestions using ShareAlbum as an example.

Simultaneously, students were given reading assignments to go through the other vulnerabilities in the lists. Students were required to read through the description, common consequences, likelihood of exploit, demonstrative examples, and potential mitigations sections for each vulnerability on the CWE website. They were also required to study the ten most critical web application security risks, especially the latest OWASP Top 10 (OWASP, 2017). Students picked two vulnerabilities from the lists (not include the three presented) and did a 10 minutes presentation to explain them.

The three common vulnerabilities we picked were cross-site scripting (CWE-79), SQL injection (CWE-89), and Unrestricted Upload of File with Dangerous Type (CWE-434).

For example, the Unrestricted Upload of File with Dangerous Type (CWE-434) was taught. After we introduced this vulnerability description, they used ShareAlbum to explain the detail of this vulnerability in practice. The vulnerable code example is shown in Figure 3. When users uploaded their pictures or videos, the code does not set restrictions on the file types, as shown in Figure 3, line 7. It created a vulnerability categorized as “Unrestricted upload of file with dangerous type.” We then demonstrated to students an attack scenario that, without restrictions on the upload file type, attackers may use this vulnerability to upload or transfer malicious executable files, which could be automatically processed within the product's environment.

```
2  function upload_image($image_file,$album_id,$location,$about,$description){
3      $album_id=(int)$album_id;
4      mysql_query("INSERT INTO photo VALUES ('','$description','$location', '".$_SESSION['user_id']."','$album_id
       ',FROM_UNIXTIME(UNIX_TIMESTAMP()),'$about')");
5      $image_id=mysql_insert_id();
6      echo $image_file, '<br />',$album_id, '<br />';
7      move_uploaded_file($image_file,'uploads/'.$album_id.'/'.$image_file);
8      Thumbnail('uploads/'.$album_id.'/', $image_file, 'uploads/thumbs/'. $album_id.'/');
9  }
```

Figure 3 CWE-434 vulnerable code in ShareAlbum.

For this vulnerability, we provided students two suggested mitigation solutions. 1) Creating an array to set the acceptable extensions. When the upload operation is processed, the restriction will be checked. If the restriction is not met, user operation is rejected. As shown in Figure 4, line 20, in ShareAlbum, developers set allowed extensions (jpg, jpeg, png, and gif). If the uploading file's extension is not in the allowed extensions, an error message "File type not allowed" will be displayed (Figure 4 line 30-31). 2) Set a limitation for the upload file size, as shown in Figure 4, line 33-34.

```
16 if(isset($_FILES['image'],$_POST['album_id'])){
17     $image_name= $_FILES['image']['name'];
18     $image_size= $_FILES['image']['size'];
19     $image_temp= $_FILES['image']['tmp_name'];
20     $allowed_ext= array('jpg','jpeg','png','gif');
21     $image_ext= strtolower(end(explode('.',$image_name)));
22     $album_id=$_POST['album_id'];
23     $about=$_POST['about'];
24     $description=$_POST['description'];
25     $location=$_POST['location'];
26     $errors=array();
27     if(empty($image_name)||empty($album_id)){
28         $errors[]='something is missing';
29     }else{
30         if(in_array($image_ext,$allowed_ext)===false){
31             $errors[]='File type not allowed';
32         }
33         if($image_size>2097152){
34             $errors[]='Maximum file size is 2mb';
35         }
36         if(album_check($album_id)===false){
37             $errors[]='Could not upload to that album';
38         }
39     }
40     if(!empty($errors)){
41         foreach($errors as $error){
42             echo $error, '<br />';
43         }
44     }
45     else{
46         upload_image($image_temp, $image_ext, $album_id,$location,$about,$description);
```

Figure 4 Mitigation code of CWE-434 in ShareAlbum.

For students to practice, we provided them three source code snippets from ShareAlbum. Students were assigned an assignment to find and fix vulnerabilities in the three categories in the given source code files.

This step delivered three learning outcomes. Students were able to 1) search for vulnerabilities and mitigation techniques to identify common vulnerabilities that frequently occur in the full life cycle development of software code, 2) understand how malicious users could make use of the three picked vulnerabilities to attack web applications, and 3) find and fix errors by examining source code for cross-site scripting errors, SQL injection errors, and missing restrictions of upload files.

Step 2: Identify Vulnerabilities

The goal of this step is to teach students secure testing skills. In this step, students were assigned two projects: to manually find errors in sample files based on the code review checklist provided by us; and to use a static analysis tool to scan software and detect vulnerabilities.

This step delivers three learning outcomes: 1) apply the manual code review using the review checklist; 2) understand how static analysis tools work; and 3) apply static analysis tools to scan software, detect errors, and recognize false-positive errors detected using the RIPS tool.

Identify Vulnerabilities via Manual Code Review

Although the manual code review is time-consuming, it is essential. The manual secure code review provides insight into the risk associated with insecure code. Besides, manual code review can effectively decrease an application's security verification cost when used together with automated testing tools (Conklin et al., 2017). By learning and practicing the manual code review, students can improve the understanding of a vulnerability's relevance and the context of what is being assessed. This procedure helps students to understand and evaluate the overall risk of vulnerabilities.

In this step, the focus is to teach manual code review using OWASP Code Review Guide (Conklin et al., 2017), SEI CERT's coding standards (Long et al., 2011; Seacord, 2008, 2014; Software Engineering Institute (SEI) at Carnegie Mellon University, 2016), and books such as Writing Secure Code (Leblanc et al., 2003), and 24 Deadly Sins of Software Security (LeBlanc et al., 2010). Students were formed into groups of three and required to go through the code together. A code review checklist adapted from the OWASP Code Review Guide was provided to students to guide them through the code review process. We demonstrated the procedure to use the vulnerable code examples from the CWE website.

Based on the code review checklist, descriptions, and vulnerable code examples of the top 25 most dangerous software errors, students generated a preliminary error list with eight errors in ShareAlbum. The eight errors should be 1) CWE-22: Improper Limitation of a Pathname to a Restricted Directory, 2) CWE-79: Improper neutralization of input during web page generation, 3) CWE-89: Improper neutralization of special elements used in an SQL command, 4) CWE-200: Information exposure 5) CWE-20: Improper input validation, 6) CWE-434: Unrestricted Upload of File with Dangerous Type, 7) CWE-798: Use of Hard-coded Credentials and 8) CWE-287: Improper Authorization.

Identify Vulnerabilities using Static Analysis Tools

Using static analysis tools is a common practice in the industry. Static analysis tools provide a convenient and scalable way to find vulnerabilities. However, they produce many false-positive cases and may miss security errors (false-negatives). We taught students to recognize the false-positives generated by the static analysis tools in this step.

To facilitate learning material adoption, a free, open-source static analysis tool, RIPS, was selected. RIPS could detect vulnerabilities by tokenizing and parsing all source code files, then detecting potentially vulnerable functions tainted by malicious users (RIPS - A static source code analyzer for vulnerabilities in PHP scripts, 2017). In the lectures, the tool usage and its pros and cons were discussed.

Students used RIPS to scan the code and generate the raw error list. They were required to submit a report about false-positives, false-negatives, and actual vulnerabilities. We provided instructions and a recorded video to guide students to prepare their environment for this project. Students were required to install their environment- PHP (WAMP or XAMPP) and RIPS. A manual and a video showing the steps to launch a static analysis scan and explain the information of RIPS discovered vulnerabilities were provided to students. They were guided to 1) download the ShareAlbum source code from the course website; 2) run RIPS from localhost using WAMP or XAMPP to conduct the first code scan; 3) input the local PHP source code location in the path/file textbox in RIPS, as shown in Figure 5; 4) choose "untainted" in verbosity level and "All" in vulnerability type, and 5) scan the code.

For the ShareAlbum program, students discovered 350 vulnerabilities using RIPS. Seven categories of errors were founded, as shown in Figure 6. The seven categories that matched the vulnerabilities categorized by CWE were 1) CWE-583 File Disclosure, 2) CWE-829 File Inclusion, 3) CWE-73 File Manipulation, 4) CWE-89 SQL Injection, 5) CWE-79 Cross-Site Scripting, 6) CWE-443 HTTP-response Splitting, and 7) CWE-470 Reflection Injection.

Figure 5 Scan setting in RIPS.

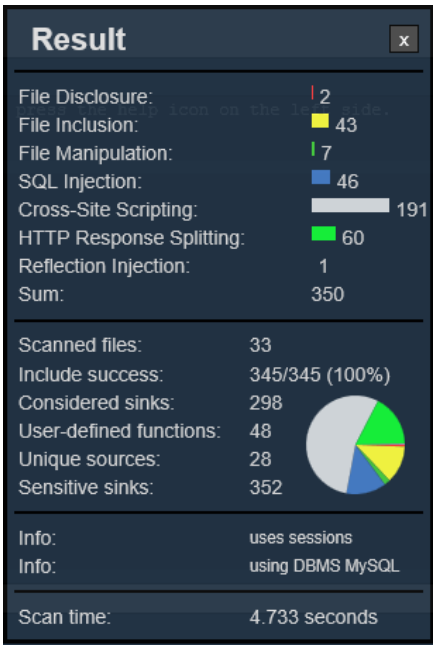


Figure 6 RIPS scan result.

We picked two false-positive errors and one false-negative vulnerability to demonstrate as examples. Students learned how to recognize false positives and remove the false positive errors from the scanned result before moving into the next step.

One example was File inclusion (CWE-829). The definition of file inclusion is "Inclusion of Functionality from Untrusted Control Sphere." File inclusion error happens when tainted user data is used to create a file name. This file name is used in an include statement. Usually, this error is detected in the HTTP GET function. It is used in "include" statement (e.g. include ("includes/" . $_GET["file"]);). The code section detected by RIPS shown in figure 7. The "include" statement does not use user-submitted data from $_GET. Thus this error is false-positive.

```
<?php
 include include ("func/comment.func.php");
?>
```

Figure 7 File inclusion error discovered by RIPS.

We demonstrated to students this false positive alarm of file inclusion detected by RIPS. To complete this step, students went through the errors discovered by RIPS and report three false positives.

Step 3: Prioritizing Vulnerabilities

Mitigation of all vulnerabilities requires too much resource, human labor, and time in commercial software development. Due to the resource limitations and deadlines, it is not practical to fix all the vulnerabilities. In this step, we taught students to focus on the most severe and high-priority issues. Other vulnerabilities with lower prioritizing scores were suggested to be documented for the next iteration.

We introduced the Common Vulnerability Scoring System (CVSS) to students. CVSS, developed by the National Infrastructure Advisory Council (NIAC), is a standard and easy-to-use system. It calculates the severity of a vulnerability (National Institute of Standards & Technology, 2019). CVSS is widely adopted to rank security errors. A CVSS score is included in almost all known vulnerabilities in the U.S. National Vulnerability Database (NVD) (National Institute of Standards & Technology, 2019). The 2020 version of the CWE Top 25 coding errors is based on the average CVSS scores and NVD counts to calculate the overall score. A vital strength of the CVSS scoring system is its simplicity. CVSS scores are computed using the CVSS score calculator. Besides, NVD provides a free online CVSS score calculator(National Institute of Standards & Technology, 2019).

In this learning step, we introduced various known vulnerabilities and their CVSS score calculation. We demonstrated how to rank security errors and manually calculated CVSS scores based on the formula's metrics. To further help students understand the CVSS metrics, we explained how to use the CVSS user guide and apply the CVSS metrics on cross-site scripting (CWE-79), SQL injection (CWE-89), and Unrestricted Upload of File with Dangerous Type (CWE-434) errors in ShareAlbum.

In the group meetings, students discuss various metrics using the CVSS score calculator and label the discovered vulnerabilities as "low," "medium," "high," and "critical" severity based upon the CVSS score. They discussed the exploitability metrics, impact metrics, temporal score metrics, environmental score metrics for each error. By manually refining the metrics, students ran the CVSS calculator to calculate the base scores, temporal scores, environmental scores, and overall scores for the vulnerabilities they discovered. Based on the CVSS overall score, students prioritized the errors and decided the top three errors to fix in the next step. They were required to submit a report about the metrics, scores of vulnerabilities, and their top three errors.

This step delivered four learning outcomes. Students should be able to 1) understand the need for the CVSS and CVSS calculation to prioritize weaknesses and vulnerabilities, 2) be familiar with the CVSS and can perform a step-by-step calculation of multiple vulnerabilities, 3) calculate a CVSS score for a newly discovered vulnerability, and 4) prioritize multiple vulnerabilities and create their own top N list.

Step 4: Mitigation

Procedures

In this step, we taught students how to fix the vulnerabilities using the existing resources. We started by asking students to find mitigation suggestions from the CWE and OWASP websites. The CWE website specifies potential mitigations for each categorized vulnerability. The OWASP top 10 list describes mitigation suggestions for each categorized vulnerability. Besides, we demonstrated remediation suggestions provided by static analysis tools (e.g., RIPS). We advised students to check the mitigation suggestions provided by static analysis tools first. Then, students went through the details of the mitigation strategies.

We taught students how to perform remediation via a step-by-step demonstration using the three vulnerabilities as examples. The three vulnerabilities we picked to demonstrate mitigation strategies in the lecture were cross-site scripting (CWE-79), SQL injection (CWE-89), and Unrestricted Upload of File with Dangerous Type (CWE-434). After students generated their own top three list in the previous learning step, they practiced mitigation approaches by making appropriate changes. They discussed the strategies in their group meeting. Then, they applied changes to the original code. Students who used a static analysis tool were suggested to scan the source code package again, seeing if they missed some vulnerabilities or made other vulnerable codes after applying the remediation code. They could go back to step two if they found vulnerabilities.

This step delivered three learning outcomes: 1) the procedure to find remediation code examples and mitigation strategy suggestions on the CWE website and OWASP top 10 list; 2) fix the errors using the CWE website's strategies; and 3) use a static analysis tool (RIPS).

Remediation Example

One of the vulnerabilities we picked to demonstrate was SQL injection (CWE-89). We introduced the description of SQL injection as following.

"SQL injection vulnerability means improper neutralization of special elements used in an SQL command. If an application developed incorrectly neutralizes special elements in SQL command, attackers could modify the intended SQL command when sent to a downstream component. It may lead to a data breach, data loss, even data modified by a malicious user."

We illustrated a piece of code to students, as shown in Figure 8. It uses echo back notifications to a user with the "*user_id*." The expected execution result should look like Figure 1a. We also explained the attack mechanism - an attacker may inject a malicious script, as shown in Figure 9. This attack produces a SQL query, as shown in Figure 10. Then we demonstrated the execution with the malicious SQL command injected. Students observed that an attacker could get all notifications with no privileges required.

```php
function get_notifys($connection){
    $notifys= array();
    $notifys_query=mysqli_query($connection,'
    SELECT notify.NotifyID,From_user.Fname,notify.Message,notify.createTime
    FROM notify,user as From_user
    WHERE notify.FromUser=From_user.userID
    AND notify.RN=0
    AND notify.ToUser='.$_SESSION['user_id']
    );

    while($notifys_row=mysqli_fetch_assoc($connection,$notifys_query)){
        $notifys[]=array(
            'notifyID'=> $notifys_row['NotifyID'],
            'From' => $notifys_row['Fname'],
            'createTime'=> $notifys_row['createTime'],
            'message'=> $notifys_row['Message']
        );
    }
    return $notifys;
}
```

Figure 8 CWE-89 vulnerable code example in ShareAlbum.

```php
<?php
$_SESSION['user_id']=$_SESSION['user_id'].' OR 1';
?>
```

Figure 9 SQL injection attack on ShareAlbum.

```sql
SELECT notify.NotifyID,From_user.Fname,notify.Message,notify.createTime
    FROM notify,user as From_user
    WHERE notify.FromUser=From_user.userID
    AND notify.RN=0
    AND notify.ToUser='.$_SESSION['user_id']
            OR 1
```

Figure 10 SQL injection result query.

For the remediation of vulnerabilities using the static analysis tool, we pointed out that students could get the remediation suggestion of a vulnerability by just clicking the error name in the scan results. RIPS listed out the files with the vulnerabilities, as shown in Figure 11. We demonstrated that students could check the error's technical details by clicking the question mark on the left-hand side. We also explained each technique details as shown in Figure 12, which includes a simple vulnerable code example, an explanation of the possible attack, and a patch section introducing suggestions to remediate the vulnerability. Then they guide students to the CWE website for more details about prevention and mitigation strategies on architecture, design, operation, and implementation.

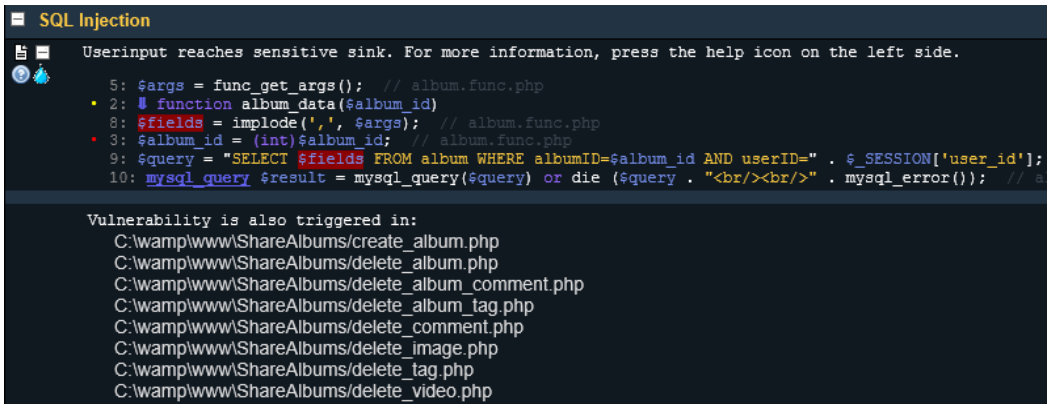


Figure 11 A vulnerability that is susceptible to the SQL injection attack.

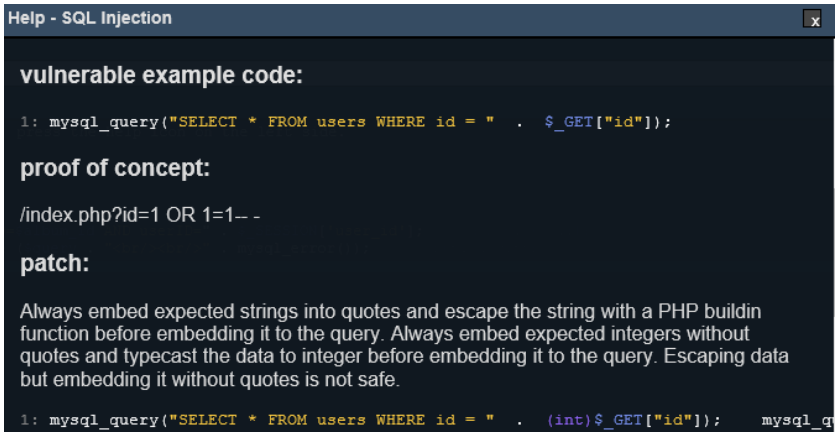


Figure 12 Technique Details of SQL injection.

We also introduced a popular remediation strategy for this kind of vulnerability - parameterization and explained the remediation code for the SQL injection error, as shown in figure 13. We introduced parameterization functions and database programming functions in PHP. For example, "*mysqli_prepare*" helps prepare SQL queries with question marks, and "*bind_param*" binds variables. We introduced another choice for this kind of error - an "accept known good" input validation strategy. Using the vulnerable code in ShareAlbum as example, we demonstrated the remediation code to modify *$_SESSION['user_id']* to *intval ($_SESSION ['user_id'])* and thus convert the session value stored in user_id to an integer. The new code rejects any input that does not strictly conform to specifications.

```
function get_notifys(){
    $notifys= array();
    $notifys_query=mysqli_prepare("
    SELECT notify.NotifyID,From_user.Fname,notify.Message,notify.createTime
    FROM notify,user as From_user
    WHERE notify.FromUser=From_user.userID
    AND notify.RN=0
    AND notify.ToUser=?"
    );
    $notifys_query->bind_param("s",$_SESSION['user_id']);
    $notifys_query->excute();
    $result=$notifys_query->get_result();
    while($notifys_row=$result->fetch_assoc()){
        $notifys[]=array(
            'notifyID'=> $notifys_row['NotifyID'],
            'From' => $notifys_row['Fname'],
            'createTime'=> $notifys_row['createTime'],
            'message'=> $notifys_row['Message']
        );
    }
    return $notifys;
}
```

Figure 13 Mitigation code of CWE-89 in ShareAlbum.

Step 5: Documentation

To integrate secure coding into the security software development cycle, companies often use standard report templates. Standard templates allow the management and security experts to direct employees to follow. We created a template by adapting the OWASP secure coding report items and the MITRE secure code review sample.

First, we introduced the OWASP secure coding report items and the sample secure code review reports published by MITRE. The OWASP standard report template classifies and prioritizes the software vulnerabilities(Conklin et al., 2017). Reports usually include the statistics data that a review team may evaluate by categories and risk levels. The MITRE secure code review samples suggest that the CWE category, source file, line number, description, and qualitative risk rating should be reported for each discovered vulnerability (MITRE, 2014).

Students were required to submit their final project report using the template, as shown in Figure 14. The template included nine items: 1) date of review, 2) application name, 3) code modules reviewed, 4) developers and code reviewer names, 5) code review checklist used, 6) static analysis tool used, 7) discovered vulnerabilities (error list without false-negative errors), 8) the top N list, and 9) discovered vulnerabilities (top three). For each vulnerability in their top three, students were asked to report, a) name of the vulnerabilities, b) description of the vulnerabilities, c) related code module and functionalities, d) source code file and line numbers, e) CVSS score, f) resolved or not, and g) remediation strategy. Table 1 provides an example of how students report a discovered vulnerability.

SECURE CODE REVIEW REPORT

- Date of review
- Application name
- Code modules reviewed
- Developers and code reviewer names
- Code review checklist used
- Static analysis tools used
- Discovered vulnerabilities (Raw error list without false negative errors)
- Top N list
- Discovered vulnerabilities (Top three)
    - Name of the vulnerabilities
    - Description of the vulnerabilities.
    - Related code module and functionalities
    - Source code file and line numbers
    - CVSS score
    - Resolved or not
    - Remediation strategy

Figure 14 Secure coding report template.

Table 1 Example of a Discovered vulnerability.

Name of the vulnerabilities	**Cross-site Scripting (CWE-79)**
Description of the vulnerabilities.	The cross-site scripting vulnerability means the improper neutralization of input during web page generation.
Related code module and functionalities	View Album, display the album ID
Source code file and line numbers	View_album.php, line 14~17
CVSS score	8.8
Solved	YES
Remediation Strategy	Check the pattern of album_id. Album_id should have been numerical, and the length of album_id should not be more than ten digits.

STUDENT FEEDBACK

About 25-35 students participated in the survey each year. We handed out a pre-survey before the training. After they submitted their reports, we asked them to complete a post-survey. We wanted to evaluate whether the five-step procedure would encourage students to apply secure coding techniques and motivate them to consider security issues in their implementation. We also wanted to study students' attitudes on this step-by-step training procedure. The study was approved by the university's Institutional Review Board (IRB).

In the first year, we taught students the secure coding process and told them the industry best practices. We had not developed the step-by-step guide by then. Students were asked to fix coding errors. Only a few students did very well.

We developed a step-by-step guide of manual code review and fixed coding errors in the second year. We obtained responses from 29 participants, with ages ranging from 19 years to 45 years, with a median age of 27 years. There were 21 male and 8 female students. About 18 (62%) of them had more than two years' coding experiences. Participants said that they were familiar with the following programming languages, C++ (25 students), C (24 students), Java (20 students), SQL (20 students), Python (13 students), JavaScript (13 students), PHP (6 students), and Ruby (4 students).

The study results showed that students understood coding errors very well (average 4.42 out of 5), and the step-by-step guide helped them prioritize and fix errors (average 4.11). Students liked how CWE/SANS Top 25 most dangerous software errors were introduced (average 4.34).

In the third year, we developed a guide using a static analysis tool (RIPS) to find and fix coding errors. Thirty students who participated in the study were between 19 and 55 years, with a median age of 29. There were 21 males and 6 females (2 students preferred not to disclosure gender information). As shown in Table 2, most participants had similar software development experiences as the previous year. For programming languages, they were familiar with C (17 students), C++ (20 students), Java (17 students), SQL (15 students), Python (15 students), JavaScript (5 students), PHP (4 students), and Ruby (1 student).

Table 2 Students' Software Development Experience.

Software development experience	Students performed the five steps on manual code review	Students performed the five steps using the static analysis tool
No experience	7	5
Half-year	2	6
One year	2	4
Two years	7	4
Three years	5	1
More than four years	6	9

Figure 15 shows the differences in students' attitudes before and after the hands-on projects. The result is encouraging- after training 51 students (more than 86% of participants) would get a list of software errors in their source code in future development vs. 33 students (about 57%) before training. In addition, after the training, more students would fix security errors in their source code than before the training (48 students, 82% after training vs. 33 students, 56% before training). Also, after training more students would document the security errors and the migration method (50 students, 86% vs. 38 students before training, 65%).

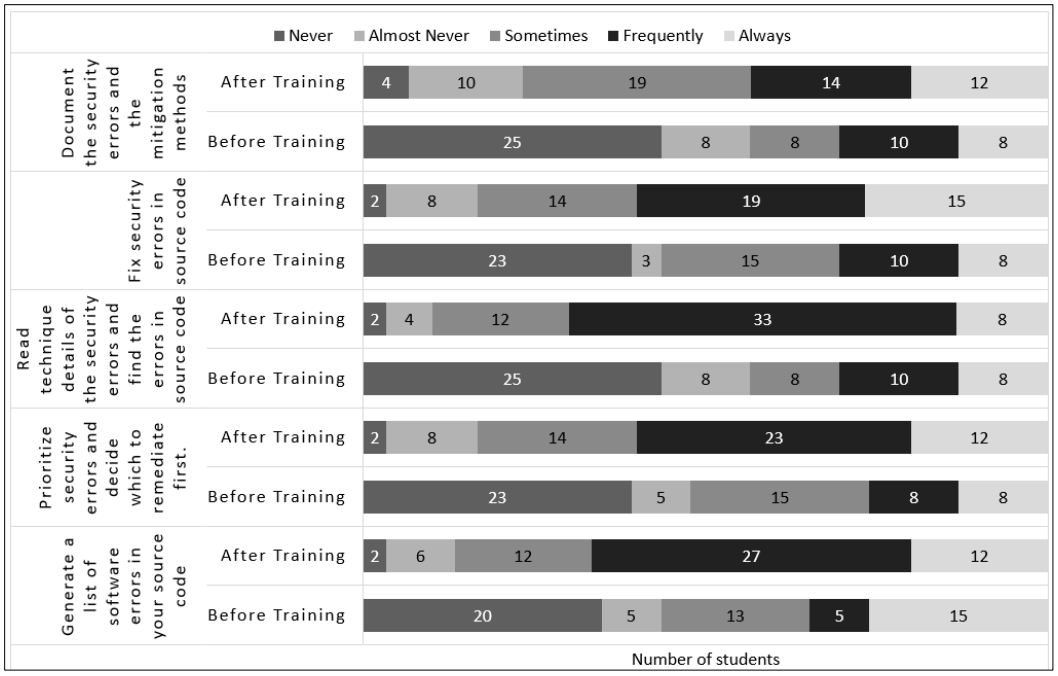


Figure 15 Comparison of the students' attitudes before and after the training.

In the post-survey, participants were asked to rate the learning material. The survey results are shown in Figure 16. It is encouraging that 20 out of 29 participants liked how we introduced secure coding projects and introduced the static analysis tool. About 40 out of 57 students like how we taught the CWE/SANS top 25 most dangerous software errors. About 44 participants were satisfied with the five steps learning procedure. Fifty of them preferred the case study using ShareAlbum. About 54 participants believed that they were satisfied with the vulnerability examples in the learning modules.

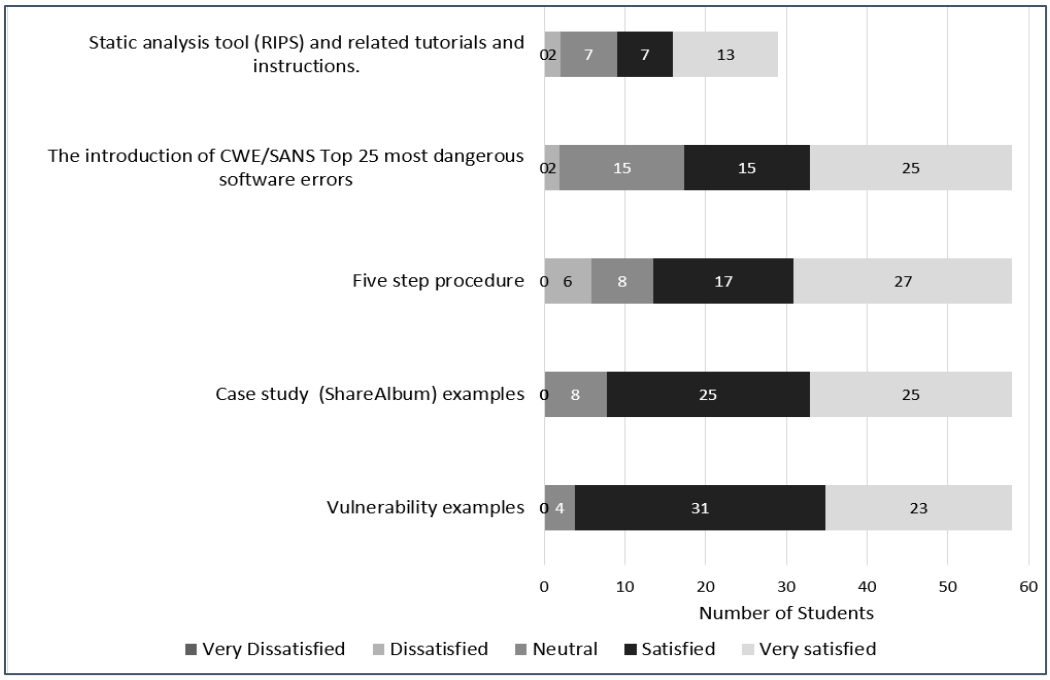


Figure 16 Participants ratings on our learning steps and materials

The learning module motivated participants to fix security vulnerabilities in their source code. Before training, only 9 participants thought that they would fix security errors in their source code. After training, 17 participants expressed they would remediate security errors in their code. As shown in Table 3, the training also significantly enhanced participants' frequencies on checking research resources about security vulnerabilities, targeting security errors, and prioritizing their secure coding vulnerabilities.

Table 3 Students' feedback before and after training.

	Before training	After training
Find vulnerabilities in source code	10	19
Check secure coding resources for security vulnerabilities	9	20
Fix security vulnerabilities in source code	9	17
Prioritize security vulnerabilities in source code	8	17

In summary, the training increased participants' motivation to perform secure software developing steps and use static tools to detect security errors. After training, participants were aware of secure coding and willing to fix security issues. Also, students liked the step-by-step guide and case studies.

CONCLUSION AND FUTURE WORK

In this paper, we proposed a five-step secure coding training approach. This approach guided students in learning common vulnerabilities, identifying vulnerabilities, prioritizing fixes, mitigating errors, and documenting the results. We provided a web application as a secure coding playground to help students practice the learning steps. In the learning steps, we presented examples of vulnerable code for common vulnerabilities. We also explained attack scenarios and mitigation suggestions.

We introduced both manual code review and static analysis using RIPS to students. By practicing the step-by-step approach in the case study, students learned the big picture and industry best practices of secure coding. They understood the common vulnerabilities and steps to discover vulnerabilities and remediation methods.

The step-by-step approach converts the complicated security errors targeting and mitigation process into small and easy-to-follow steps. This approach facilities the adoption of industry best practices and secure coding skills. The students' feedbacks show that they were more motivated to fix security vulnerabilities and interested in secure software development. Furthermore, students like to use secure coding resources and automatic tools to solve security-related issues. Students learned and practiced secure skills in the learning steps when mitigating the most common vulnerabilities. We taught secure software development using the best industry practices and relative resources. Students' feedbacks indicated that the five-learning steps are efficient ways to educate secure software development.

Future research is needed to address the following questions. First, why students frequently conduct manual code reviews versus static analysis tools (20 vs. 16). Second, what are the fundamental reasons students perform differently; some can fix errors quickly, while others take a long time and fail. Our ongoing research uses eye-tracking devices to study students' behavior during the secure coding exercises. Third, we are further improving and investigating learning procedures by developing more learning activities and investigating hands-on projects using dynamic analysis tools. In addition, to improve this step-by-step approach, we are in the process of updating the learning modules, hands-on projects, and designing new case studies in different programming languages.

References

Agnitio - Static analysis. (2015). Retrieved from https://sourceforge.net/projects/agnitiotool/

Arciniegas, F., Bartoldus, M., Carter, J., Challey, D., Chess, B., Clarke, J., Cornell, D., Craigue, M., Cruz, D., Deleersnyder, S., Derry, J., De Win, B., Dickson, J., Fakos, A., Fern, D., Glas, B., Hinojosa, K., Hoff, J., Huth, C., … Wierckx, S. (2019). Software Assurance Maturity Model A guide to building security into software development OWASP The Open Web Application Security Project. *Owasp*, 1–72.

Chen, L., Tao, L., Li, X., & Lin, C. (2010). A tool for teaching web application security. *Proceedings of the 14th Colloquium for Information Systems Security Education*, 17–24.

Chu, B., Stranathan, W., Cody, J., Peterson, J., Wenner, A., & Yu, H. (2009). Teaching secure software development with vulnerability assessment. *Proceedings of the 13 Colloquium for Information Systems Security Education (CISSE 2009). Seattle, Washington*.

Cisco. (2016). *Secure Development Lifecycle*. Retrieved from https://www.cisco.com/c/dam/en_us/about/doing_business/trust-

center/docs/cisco-secure-development-lifecycle.pdf

Coley, S. C. (2014). *Common Weakness Scoring System*. Retrieved from https://cwe.mitre.org/cwss/cwss_v1.0.1.html

Conklin, L., & Robinson, G. (2017). *OWASP Code Review Guide, V2.0*. Owasp. Retrieved from https://owasp.org/www-project-code-review-guide/

CVSS. (2020). *CVSS Severity Distribution Over Time*. Retrieved from https://nvd.nist.gov/general/visualizations/vulnerability-visualizations/cvss-severity-distribution-over-time

CWE. (2020). *CWE-79 Cross-site Scripting*. Retrieved from https://cwe.mitre.org/data/definitions/79.html

CWE Common Weakness Enumeration. (2014). Retrieved from http://cwe.mitre.org/

CWE List. (2020). Retrieved from https://cwe.mitre.org/data/index.html

CWE Top 25 Most Dangerous Software Weaknesses. (2020). Retrieved from https://cwe.mitre.org/top25/archive/2020/2020_cwe_top25.html

Dodson, D., Souppaya, M., & Scarfone, K. (2019). Mitigating the Risk of Software Vulnerabilities by Adopting a Secure Software Development Framework (SSDF). In NIST.

Du, W., Teng, Z., & Wang, R. (2007). SEED: a suite of instructional laboratories for computer security education. *ACM SIGCSE Bulletin*, *39*(1), 486–490.

Dukes, L., Yuan, X., & Akowuah, F. (2013). A case study on web application security testing with tools and manual testing. *Southeastcon, 2013 Proceedings of IEEE*, 1–6.

Kaza, S., Taylor, B., Hochheiser, H., Azadegan, S., O'Leary, M., & Turner, C. F. (2010). Injecting security in the curriculum--experiences in effective dissemination and assessment design. *The Colloquium for Information Systems Security Education (CISSE)*, *Volume 8*.

Leblanc, D., & Howard, M. (2003). *Writing Secure Code*. Pearson Education.

LeBlanc, D., & Viega, J. (2010). *24 deadly sins of software security: programming flaws and how to fix them*. McGraw-Hill.

Long, F., Mohindra, D., Seacord, R. C., Sutherland, D. F., & Svoboda, D. (2011). *The CERT Oracle Secure Coding Standard for Java*. Addison-Wesley Professional.

Meucci, M., & Muller, A. (2013). *OWASP Testing Guide 4.0*. Retrieved from https://owasp.org/www-pdf-archive/OTGv4.pdf

Microsoft. (2012). *Microsoft Security Development Lifecyle Version 5.2*. Retrieved from https://docs.microsoft.com/en-us/previous-

versions/windows/desktop/cc307748(v=msdn.10)

Microsoft. (2016). *Microsoft Threat Modeling Tool*. Microsoft. Retrieved from https://www.microsoft.com/en-us/download/details.aspx?id=49168

Microsoft. (2018). *Enable or install first-party .NET analyzers*. Retrieved from https://docs.microsoft.com/en-us/visualstudio/code-quality/install-fxcop-analyzers?view=vs-2019#nuget-package

Microsoft. (2020a). *Microsoft SDL practices*. Retrieved from https://www.microsoft.com/en-us/securityengineering/sdl/practices

Microsoft. (2020b). *Secure coding guidelines in .NET*. Retrieved from https://docs.microsoft.com/en-us/dotnet/standard/security/secure-coding-guidelines

MITRE. (2014). *Sample Secure Code Review Report*. Retrieved from http://www.mitre.org/sites/default/files/publications/secure-code-review-report-sample.pdf

MITRE. (2020a). *Common Attack Pattern Enumeration and Classification*. MITRE. Retrieved from http://capec.mitre.org/data/

MITRE. (2020b). *Common Vulnerabilities and Exposures*. Retrieved from https://cve.mitre.org/

National Initiative for Cybersecurity Careers and Studies. (2020). *NICE Cybersecurity Workforce Framework*. Retrieved from https://niccs.cisa.gov/workforce-development/cyber-security-workforce-framework

National Institute of Standards & Technology. (2019). *Common Vulnerability Scoring System Version 3.1*. Retrieved from https://nvd.nist.gov/vuln-metrics/cvss/v3-calculator

NIETP. (2020). *National Center of Academic Excellence Cyber Defense Education Knowledge Units*. Retrieved from https://www.iad.gov/NIETP/documents/Requirements/CAE-CD_2020_Knowledge_Units.pdf

OWASP. (2020). *Vulnerability Scanning Tools*. Retrieved from https://owasp.org/www-community/Vulnerability_Scanning_Tools

OWASP Development Guide. (2005). Retrieved from https://www.owasp.org/index.php/Projects/OWASP_Development_Guide

OWASP LAPSE+ Static Code Analysis Tool for Java. (2017). Retrieved from https://wiki.owasp.org/index.php/OWASP_LAPSE_Project

OWASP ZAP. (2020). OWASP. Retrieved from https://www.zaproxy.org/

Pérez, P. M., Filipiak, J., & Sierra, J. M. (2011). LAPSE+ static analysis security software:

Vulnerabilities detection in java EE applications. In Future Information Technology (pp. 148–156). Springer.

Purdue University. (2018). *CS 52700 - Software Security*. Retrieved from https://catalog.purdue.edu/preview_course_nopop.php?catoid=8&coid=82319

Pylint - python code analysis tool. (2020). Retrieved from https://www.pylint.org/

RIPS - A static source code analyzer for vulnerabilities in PHP scripts. (2017). Retrieved from http://rips-scanner.sourceforge.net/#screenshots

Rothke, B. (2006). 24 Deadly Sins of Software Security. In Security Management (Vol. 50, Issue 2).

SAFECode. (2018). *Fundamental Practices for Secure Software Development*. *March*, 38.

Seacord, R. C. (2005). *Secure Coding in C and C++*. Pearson Education.

Seacord, R. C. (2008). *The CERT C secure coding standard*. Pearson Education.

Seacord, R. C. (2014). *The CERT C coding standard: 98 rules for developing safe, reliable, and secure systems*. Pearson Education.

Shostack, A. (2014). *Threat modeling: Designing for security*. John Wiley & Sons.

Software Engineering Institute (SEI) at Carnegie Mellon University. (2016). *SEI CERT C++ Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems*. Retrieved from https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=494932

Software Engineering Institute (SEI) at Carnegie Mellon University. (2020). *Curricula: Software assurance Materials and Artifacts*. Retrieved from https://www.sei.cmu.edu/education-outreach/curricula/

Software Engineering Institute (SEI) at Carnegie Mellon University. (2021). *Curricula: Software assurance Materials and Artifacts*. Software Engineering Institute (SEI) at Carnegie Mellon University. Retrieved from https://www.sei.cmu.edu/education-outreach/curricula/

Sullivan, B., Bonver, E., Furlong, J., & Orrin, S. (2013). *Practices for Secure Development of Cloud Applications*. SAFECode & Cloud Security Alliance. Retrieved from https://safecode.org/publication/SAFECode_CSA_Cloud_Final1213.pdf

Taylor, B., & Kaza, S. (2011). Security injections: modules to help students remember, understand, and apply secure coding techniques. *Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education*, 3–7.

The Owasp Foundation. (2010). *OWASP Secure Coding Practices Quick Reference Guide*. OWASP. Retrieved from https://owasp.org/www-pdf-archive/OWASP_SCP_Quick_Reference_Guide_v2.pdf

Towson University. (n.d.). *CLARK*. Retrieved from https://clark.center/home

Towson University. (2020). *Security Injections*. Towson University. Retrieved from http://cis1.towson.edu/~cyber4all/index.php/security-injections_home/

Veracode. (2020a). *Veracode Dynamic Analysis*. Veracode. Retrieved from https://www.veracode.com/products/dynamic-analysis-dast

Veracode. (2020b). *Veracode Static Analysis*. Veracode. Retrieved from https://www.veracode.com/products/binary-static-analysis-sast

Walden, J., & Doyle, M. (2012). SAVI: Static-Analysis vulnerability indicator. *IEEE Security and Privacy*, *10*, 32–39. doi: 10.1109/MSP.2012.1

Walden, J., & Frank, C. E. (2006). Secure software engineering teaching modules. *Proceedings of the 3rd Annual Conference on Information Security Curriculum Development*, 19–23.

Wenliang Du. (2020). *SEED lab*. Retrieved from https://seedsecuritylabs.org/

Wheeler, D. A. (2017). *Flawfinder*. Retrieved from https://dwheeler.com/flawfinder/

Whitney, M., Lipford, H. R., Chu, B., & Thomas, T. (2018). Embedding secure coding instruction into the ide: complementing early and intermediate CS courses with ESIDE. *Journal of Educational Computing Research*, *56*(3), 415–438.

Xiaohong Yuan. (2019). *Secure Coding*. Retrieved from https://clark.center/details/xhyuan/7bd8a138-4f36-45f7-acd7-dfae0a6691cf

Xie, T., Bishop, J., Tillmann, N., & De Halleux, J. (2015). Gamifying software security education and training via secure coding duels in code hunt. *Proceedings of the 2015 Symposium and Bootcamp on the Science of Security*, 1–2.

Yuan, X., Yu, H., Hernandez, J., & Wadell, I. (2012). Integrating software security education into computer science curriculum. *Proc. of the 11th IASTED International Conference on Software Engineering*.

Zeng, M., & Zhu, F. (2020). *Secure software development*. Retrieved from https://sites.google.com/a/uah.edu/pervasive-security-privacy/secure-software-development