A Step by Step Guide to Building Secure Software

Mini ZENG
Department of Computer Science, the University of Alabama in Huntsville
Huntsville, AL 35899, USA

and

Feng ZHU
Department of Computer Science, the University of Alabama in Huntsville
Huntsville, AL 35899, USA

ABSTRACT

Software vulnerabilities have become widespread in recent years. Multiple research organizations have published common software security errors. It is, however, overwhelming for developers to read through the long documentations. There are few studies on how to educate developers use these research resources. We suggest a step-by-step approach for programmers and designers to mitigate security errors. Our approach guides developers to generate an error list, prioritize errors by risk evaluations, target security errors in source code, test attack, establish mitigation strategy, and document the results. We provide a case study to illustrate the approach and established mitigation strategies by using the 2011 CWE/SANS Top 25 Most Dangerous Software Errors. We present examples, determine the priorities for the fixes, and mitigate errors. We evaluate the method by surveys and experiment. Result shows that the step by step guide and case study training could increase participants' motivation to use CWE resources and perform software security developing steps.

Keywords: Secure, Software design, Error handing and recovery.

1. INTRODUCTION

On-line applications involving financial and personal data are now in wide use. Many of these, such as online banking, auto-insurance, and health insurances, are critical and frequently become the target of attackers [1]. Even web applications that use non-confidential data can contain and spread malware. Software developers may neglect security concerns. This is common as developers are focusing on functionality. Software vulnerabilities reported by Common Vulnerabilities and Exposures (CVE) [2] from 1999 through January 2015 show a clear and pressing need for software developers to learn and incorporate secure software development practical procedures.

Research Organizations have created numerous resources for software security issues, such as Common Weakness Enumeration (CWE), the Open Web Application Security Project (OWASP), and a Software Assurance Forum for excellence in code (SAFECode). In 2009, 2010, and 2011, CWE published three versions of the CWE/SANS Top 25 Most Dangerous Software Errors. In 2014, CWE published the latest CWE report, Version 2.8, which included a thousand errors and error categories [3]. OWASP Software Assurance Maturity Model Project specified a framework for the design and development of secure software [4]. The OWASP Development Guide provides practical instructions and J2EE, ASP. NET and PHP code samples [5]. SAFEcode published two versions of secure development practices with an analysis of real-world actions [6]. SAFEcode and Cloud Security Alliance released a guide to help readers better understand and implement best practices for secure cloud applications' development [7]. CWE and OWASP, for instance, provide a large amount of resources on security errors and software vulnerabilities discovered each year. While these research resources are valuable, developers may be overwhelmed by the amount of documentation and number of error lists and not willing to use them in their practical developing. In addition, few resources are available to teach developers on how to estimate tradeoffs between mitigating security errors, and development and testing (such as time, labor, etc.).

In this paper, we propose a progressive approach and use a case study to train software developers on target and mitigate security errors. We use the 2011 CWE/SANS Top 25 Most Dangerous Software Errors as a guide. One of our major goals is to introduce a step-by-step procedure for programmers and designers to solve security-related issues in their code and design. This approach reduces complexity by splitting the whole approach into smaller, easier-to-handle processes.

To illustrate this method in a practical procedure, our discussion will focus on a simple but fully functional application named ShareAlbum. This application was developed by three students with good programming skills. Two of them were in a team that won first place in regional and the Popular Choice Award in the global competition at the America's Datafest 2013. The whole project source code can be downloaded from our website [8] for practice.

This paper makes two main contributions. First, we provide integration between research resources from CWE, teaching developers target and fix software security issues by a step by step guide and a practical example (ShareAlbum). Second, we introduce a method to prioritize security errors, which is named S-value. S-value provides an overall evaluation of remediation cost, attack frequency, ease of detection and attacker awareness. S-value can provide flexibility to software development teams by prioritizing the errors according to their situation.

The rest of the paper is organized as follows. We discuss the background and related work in Section 2. Then, we describe a method to find and fix security errors in Section 3. In Section 4, we use ShareAlbum as a case study to mitigate errors. We evaluate our method in Section 5. And finally in Section 6, we give concluding remarks.

2. BACKGROUND AND RELATED WORK

In this section, we provide background information about CWE, 2011 CWE/SANS Top 25 Most Dangerous Software Errors. We also specify tools that developers can use to detect errors and related work that OWASP and SAFEcode did in recent years.

CWE is a community developed repository of software weakness types that is useful in all stages of development and design [9]. Software managers can incorporate software weakness analysis as part of their process. Programmers can find weaknesses in their source code. And, customers can use this list to check the security of software they purchased. Researchers in software security can focus their researches toward the specific vulnerabilities enumerated within.

The 2011 CWE/SANS Top 25 Most Dangerous Software Errors is a list of the most severe and common software errors. These errors are based on more than 800 programming errors, design errors and architecture errors which could lead to serious vulnerabilities [10]. The errors are scored and ranked on the Common Weakness Scoring System (CWSS) [10].

CWE specifies weakness prevalence, consequence, level of remediation cost, ease of detection, attack frequency and attacker awareness for each error. According to CWSS rank, the highest score is given to improper neutralization of special elements used in an SQL Command ('SQL' injection). This error has high weakness prevalence, low remediation cost, easy level of detection, and high attacker awareness. The consequences of 'SQL' injection are data loss and security bypass. The second most dangerous software error is improper neutralization of special elements used in an OS command (OS command injection). This error costs more to remediate than the 'SQL' Injection error. It also has high attacker awareness and is easy to detect. Buffer overflow comes in the third. This is a widespread error that leads to malicious code execution, denial of service and data lose [10].

Detection methods and effectiveness are listed under Technique Details on the CWE website [11]. Demonstrative examples include remediation code that could help programmers understand each error in multiple language environments. The Potential Mitigation section describes mitigation strategies. The Monster Mitigations section provides guidance to software managers to help them find out approaches to alleviate security software errors in their project. Analysis of weakness prevalence, remediation cost, ease of detection, consequences, attack frequency, and attacker awareness for each error could help software managers decide which error to address first.

There are many automated tools to help programmers locate errors. Static and Dynamic analyses are two of the most popular types of security test. Static analyses tools discover security errors without running the program. Open source static analysis tools include PMD for Java, FlawFinder for C/C++, Microsoft FxCop for .NET and RIPS for PHP [12]. Coverity provides static analyses tools for C, C++, Java and C# [13]. Dynamic analysis tools examine software by executing the program and observing system memory, functional behavior, response time and overall performance [14]. Dynamic analysis tools such as QAInspect, WebInspect, HP Security Suite and IBM Security AppScan provide security solutions targeted toward different stages of the development lifecycle . Veracode provides both Static Application Security Testing and Dynamic Application Security Testing [15]. Attack surface tools like Attack Surface Analyzer(Microsoft) helps developers view changes in the attack surface resulting from the introduction of their code onto the Windows platform [16]. Fuzz testers such as Zzuf, Peach and Radamsa aim to detect errors in the program code but do not rely on previously known vulnerabilities [6]. Web application vulnerability scanners such as Nmap can provide vulnerability data, asset information and threat detection [17]. James Walden and Maureen Doyle developed an indicator named SAVI (Static-Analysis Vulnerability indicator) to evaluate web application security risks on the basis of static analysis of source code [18]. However, these tools aren't perfect. They are struggling to balance false-positive warnings, which reports defect-free problems in code and false-negative problems [19].

OWASP is a free and open software security community. Their "Top 10" is an awareness document for web application security. The latest OWASP Top 10 are: Injection, Broken Authentication and Session Management, Cross-Site Scripting, Insecure Direct Object References, Security Misconfiguration, Sensitive Data Exposure, Missing Function level Access Control, Cross-Site Request Forgery, Using Components with known Vulnerabilities, Invalidated Redirects and Forwards [20]. OWASP provides verification method, attack scenarios example and prevention method for each security risk.

SAFECode is a global non-profit organization. Their main goal is to identify and promote best practices for developing and delivering more secure and reliable software, hardware and services [21]. In 2011, SAFECode published the latest version [6] of security practices. SAFEcode best practices help developers provide stronger controls and integrity for commercial applications [6]. The best practices are applied during the design, programming and testing phases. SAFECode includes methods and tools to verify each practice, mitigation, and CWE references for each practice listed. In 2013, SAFECode provided additional secure development recommendations in the context of critical threats to cloud computing [7].

3. METHODOLOGY

Based on the CWE Top 25 Most Dangerous software Errors we developed a step-by-step approach for location and resolution of software errors. Figure 3 and following paragraphs describe each step of this solution.

Step 1 Generate raw error list. Software managers or developers should decide on using automated detection methods or manual static analysis to create a raw error list. If automated detection methods are used it will generate a list of errors after a scan. We will use this list as a raw error list. Sometimes, manual static analysis may be a more desirable solution to provide sufficient code coverage because it could reduce false-positive alarms and adapt to limited time constraints. For manual static analysis, developers should go through the brief list of Top 25 Most Dangerous Software Errors and identify potential software errors. If an error name is hard to understand, click the error ID to get a detailed description. Compare the applicable platform with language and technology of the software. Eliminate the ones which are not fit. Then establish a raw error list with potential errors.

Step 2 Risk evaluations. Developers should use the following sub-steps to evaluate each error in the raw error list.

- Check the CWE summary (red box 1 in Figure 1) for level of attacker awareness, attack frequency, remediation cost and ease of detection and then tabulate them (as shown in Table 2 in section 4).
- Discuss the errors with software managers to determine the significance and assign weights to them in a tabulated form (as shown in Table 1 section 4). For example, if budget is a main limitation, software managers should assign higher weight for remediation cost. Other constraints may also be considered. For example, if the release date is coming soon, weight on ease of detection should be more than others.
- Calculate S-value and order errors by descending S-value. Details of S-value calculations will be presented in section 4. Based on S-value, group the errors. The first group will contain errors with highest S-value and will need to be fixed immediately. The second and the third group will contain errors which will be solved in future releases. The reason that we group these errors is that in the software engineering context, developers cannot mitigate all the security errors in their first release. In most cases, software developing teams may have pressure of release date, so they will need to fix the security issues categorized as catastrophic and critical with frequently appearance in code [22].

Step 3 Errors code targeting. For each error, a developer should check code examples (red box 2 in Figure 1) with the corresponding programming language on the CWE website. Detection methods (red box 3 in Figure 1) including effectiveness will give developers a suitable method for addressing a specific error. Automatic detection tools and manual analysis tools have different solutions here.

- If an automated detection method is used, it already provides a table that includes the error line number, file name, error name and ID. Developers should select the errors from this table based on risk evaluation. False-positive errors will be eliminated by the following steps.
- If manual static analysis is used, developers should review the software document first, classify code files according to their functionalities and then identify the category of functionality that may include the error code based on code examples.
- Open a suspicious code file. Then, put the code file and code examples side by side to target the lines of code that include the security error.

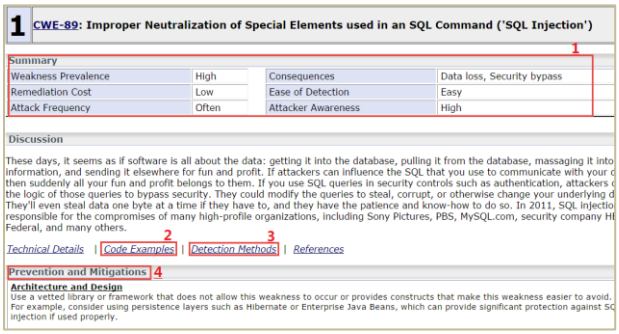


Figure 1 CWE software security error example [24].

Step 4 Attack and Mitigation. We tested each error in the processing group by attacks to help developers understand the errors. Developers don't have to test them by attacks in a practical circumstance. They should design and implement mitigation strategies for the errors. We use CWE-79 as an example to illustrate a testing attack and an implementation of mitigation strategy in section 4. (Detailed mitigation strategies can be found on the CWE website in the code example part.) More prevention and mitigation strategies on architecture, design, operation and implementation are also listed on the CWE website (red box 4 Figure 1). Developers decide mitigation approaches, make appropriate changes on the lines of code targeted in step 3 and go through all the project code to mitigate errors in the first group.

Step 5 Documentation. The development team documents the list of errors, the code files and functions which contain errors, mitigation strategies and update timestamp.

If more time and budget are available, repeat step 3, step 4 and step 5 for the second group of errors.

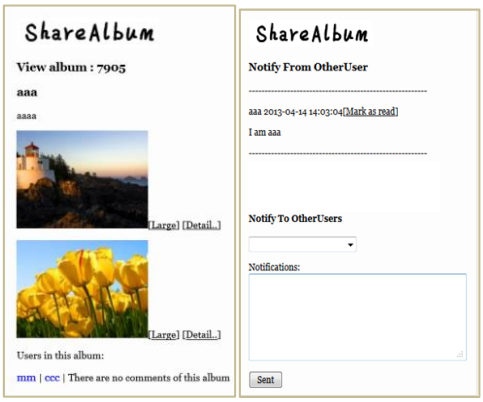
4. CASE STUDY

To illustrate the method represented in section 3, we will discover and mitigate security errors on ShareAlbum. We chose ShareAlbum because it has a fully functional application that involves classic web operations such as uploading images and videos, sending and receiving messages between users, user registration, etc. It also requires private information which users only want to share within a self-defined group. This web application was developed by experienced developers, but many typical security errors were there.

4.1 Reviewing ShareAlbum

ShareAlbum is used to share albums, photos and videos with other users. This application uses PHP, HTML and MySQL. The ShareAlbum database stores and keeps track of photos, videos, photo-tags, users' information, etc.

The albums and videos can be categorized as private or public when they are created. Members have privileges to review, make comments and tags on public photos and videos (Figure 2-a). Private photos and videos can only be reviewed by the owner. Users can send messages to each other. Users will be notified of new messages after they logged in. Every message will contain the sender's ID, message content and send time (Figure 2-b).



(a) (b)

Figure 2 Screen shots of ShareAlbum

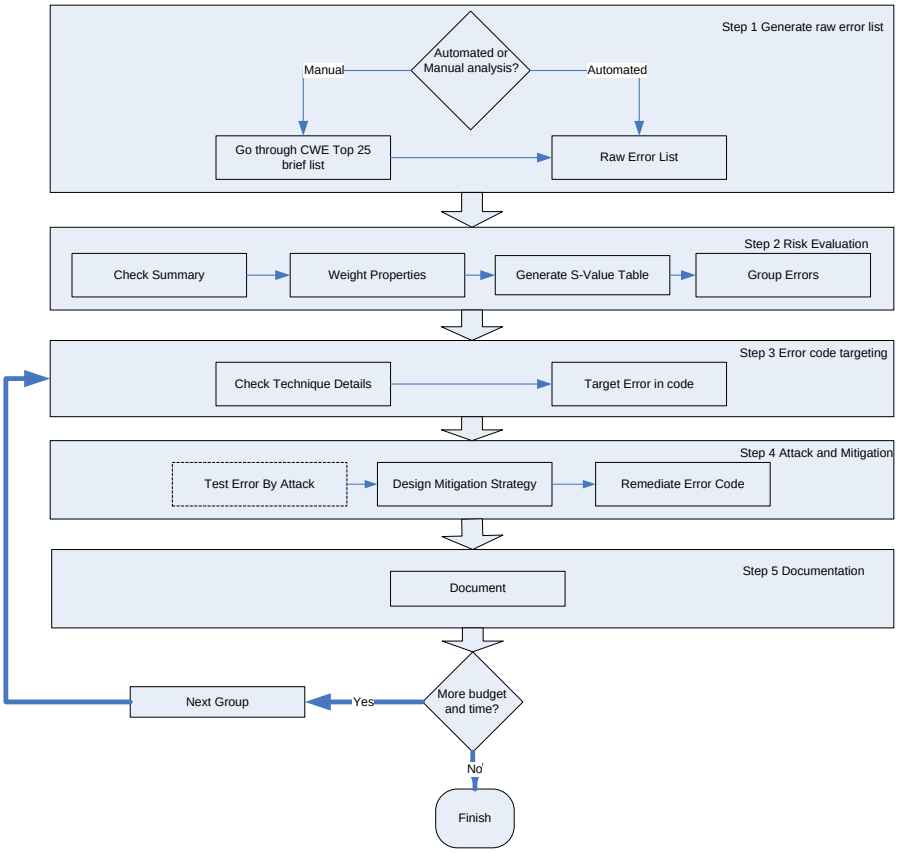


Figure 3 Path to Fix Software Security Issues

4.2 Technique Details and Mitigation

We followed the method that was described in section 3. For Step 1, we used manual static analysis. After going through the brief list of Top 25 Most Dangerous Software Errors, we generated a raw error list with nine errors: 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-311: Missing Encryption of Sensitive Data. 5) CWE-327: Use of a Broken or Risky Cryptographic Algorithm. 6) CWE-434: Unrestricted Upload of File with Dangerous Type. 7) CWE-759: Use of a One-Way Hash without a Salt. 8) CWE-798: Use of Hard-coded Credentials. 9) CWE-862: Missing Authorization. Then we checked applicable platform of them on the CWE website. Languages and architectural paradigm requirements of these nine errors are all fit to ShareAlbum.

For Step 2, we evaluated security errors' properties based on time and budget limitations. These properties are remediation cost, attack frequency, ease of detection and attacker awareness. We then calculated the values as shown in Table 1.

According to the summary of each error in the raw error list, we tabulated the level of remediation cost, attack frequency, ease of detection and attacker awareness Table 2.

Then, we calculated the S-values base on these two tables using Eq. (1).

Table 1 Value and weight for security error properties

Value / Weight	High/Often	Medium/ Moderate/ Sometimes	Low/ Easy
Remediation Cost(W_R=4)	V_R=1	V_R=2	V_R=3
Attack Frequency(W_{AF}=3)	V_{AF}=3	V_{AF}=2	V_{AF}=1
Ease of Detection(W_E=2)	V_E=1	V_E=2	V_E=3
Attacker Awareness(W_{AA}=1)	V_{AA}=3	V_{AA}=2	V_{AA}=1

Table 2 Property of each error in ShareAlbum

	Remediation Cost (R)	Attack Frequency (AF)	Ease of Detection (E)	Attacker Awareness (AA)
CWE-79	Low	Often	Easy	High
CWE-89	Low	Often	Easy	High
CWE-862	Low to Medium	Often	Moderate	High
CWE-434	Medium	Sometimes	Moderate	Medium
CWE-798	Medium to High	Rarely	Moderate	High
CWE-331	Medium	Sometimes	Easy	High
CWE-22	Low	Often	Easy	High
CWE-759	Medium to High	Rarely	Moderate	High

$$S - value = V_R \times W_R + V_{AF} \times W_{AF} + V_E \times W_E + V_{AA} \times W_{AA} \quad (1)$$

For example, CWE-79 with low remediation cost, often attack frequency, easy detection and high attacker awareness will have V_R=3, V_{AF}=3, V_E=3, V_{AA}=3 according to Table 1. While the weight of remediation cost is W_R=4, attack frequency is W_{AF}=3, ease of detection is W_E=2, and attacker awareness is W_{AA}=1. According to the above formula, we can get an S-value=30 for CWE-79. We calculated the S-value for all errors in Table 2, then listed them by S-value in descending order. The final list in our case is CWE-79 (S-value=30), CWE-89 (S-value=30), CWE-862 (S-value=26), CWE-22 (S-value=26), CWE-434 (S-value=20), CWE-331 (S-value=19), CWE-798 (S-value=16) and CWE-759 (S-value=16).Then we grouped the ones with higher S-value into the first group which consists of CWE-79, CWE-89, CWE-862 and CWE-22. Higher S-value implies relatively low remediation cost, high attack frequency, easy detection and high attacker awareness. We will test attacks and mitigate the errors in this group first. And then solve the other four if time and budget allow.

For step 3, developers may download software documentation and source code from our website. We reviewed document of ShareAlbum to target these errors in files. Software Component graph (Figure 4) could guide the manual static analysis process. Then, we maped each error with file names as shown in Table 3.

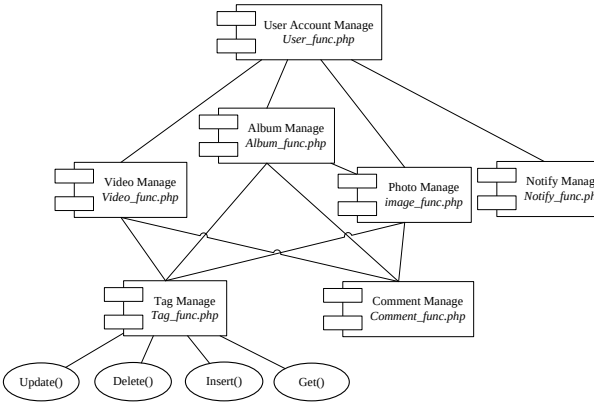


Figure 4 Component graph of ShareAlbum

For Step 4, we depicted code examples, attacks and mitigations on error CWE-79, which has the highest S-value. On our website, we provide technical details and mitigations on CWE-89, CWE-862, and CWE-22 and discuss attacks and mitigations for errors CWE-434, CWE-331, CWE-798 and CWE-759 existing in ShareAlbum.

CWE-79: Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting') [23]. Attackers could inject JavaScript or other browser executable script into a web page. When our web page was loaded by other users, their browsers could execute the malicious script attackers injected. Cross-site Scripting can be detected by static analysis and dynamic analysis. According to the discussion and description of CWE-79 on CWE website, we can conclude that cross-site errors may

Table 3 Relationship between files and errors in ShareAlbum

Errors	Files
CWE-79	View_ablum.php, view_public_album.php,
CWE-89	Notify_func.php Image_func.php Video_func.php Album_func.php
CWE_862	Album_func.php Image_func.php
CWE_434	Upload_image.php Upload_video.php
CWE_798	Init.php
CWE_311	User_func.php
CWE_22	Image_func.php Video_func.php
CWE_739	User_func.php

exist in ShareAblums code using HTTP GET parameters. And these lines of code can be found in View_ablum.php and View_public_album.php. The CWE-79 error in View_public_album.php is illustrated in Figure 5 emphasized by red underline.

```
12  include 'template/header.php';
13
14  $album_id=$_GET['album_id'];
15  $album_data=public_album_data($album_id, 'albumName', 'description');
16
17  echo '<h3>View album : '.$album_id. '</h3>';
18
19  echo '<h3>',$album_data['albumName'],'</h3><p>',$album_data['description'],'</p>';
```

Figure 5 Code example of CWE-79

These lines of code are used to view information of an album with an album id as the headline. Line 14 in Figure 5 fetched album id by using HTTP GET album_id parameter. Line 17 displays the album id by using the parameter defined in line 14. Line 19 used to show the album name and description.

The view_public_album web page without attack should be like Figure 3-a. An attacker may add the following lines in URL to embed a fake login box in the web page.

https://ShareAlums.com/view_public_album.php?album_id=%3Cdiv+id%3D%22stealPassword%22%3EPlease+Login%3A%3Cform+name%3D%22input%22+action%3D%22http%3A%2F%2Fattack.example.com%2FstealPassword.php%22+method%3D%22post%22%3EUsername%3A+%3Cinput+type%3D%22text%22+name%3D%22username%22+%2F%3E%3Cbr%2F%3EPassword%3A+%3Cinput+type%3D%22password%22+name%3D%22password%22+%2F%3E%3Cinput+type%3D%22submit%22+value%3D%22Login%22+%2F%3E%3C%2Fform%3E%3C%2Fdiv%3E%0D%0A

Once these lines are injected, the webpage loaded by a user will execute the malicious script. The result is shown in Figure 6 where the red box shows the fake login form. Once the user inputs password and username, then clicks login, this information will be sent to the attacker. Complicate JavaScript lines or other browser executable script could also be injected in this way. Plenty of mitigation strategies could be used for the Cross-site Scripting error. "Accept known good" input validation strategy is suggested in the CWE website. To mitigate the Cross-site Scripting, we should specify the variable transfer between webpages. Line 14-19 in Figure 5 should be changes to the lines in Figure 7-a. We checked the pattern of album_id. In our case, album_id should have been numerical

(mitigation code shown in Figure 7 line 16-18), and the length of album_id should not be more than 10 digits (mitigation code shown in Figure 7 line 19-24). We provided corresponding pseudo code shown in Figure 7-b to explain the lines of PHP code in Figure 7-a.

Figure 6 Attack result with fake login form

5. EVALUATION

To evaluate our method, we recruited participants from graduate level students from the computer science departments. They were doing this as a homework project. We asked the participants to read the guide and then target and mitigate the secure errors in ShareAlbum. After they submit their report, we ask them to complete a post survey. The procedures of the experiment were approved by our university's IRB.

We obtained responses from 29 participants with ages ranging from 19 years to 45 years with a median age being 27 years with 72.4% males and 27.6% females. The ethnic diversification was as follows – 48.3% of participants were Asian or Asian-American, 24.1% were Caucasian, 24.1% were categorized themselves as others, 3.4% were African or African-American and 0% of the participants were Hispanic, Latino or Mexican-America. 20.7% of participants have more than four years software development experience. 17.2% have about 3 years' experience. 24.1% have about 2 years' experience. 6.9% have about 1 year experience. 6.9% have about half years' experience. And 24.1% have non experience on software development. Participants claimed they are familiar with following programming languages, C++ (86.2%), C (82.7%), Java (68.9%), SQL (68.9%), Python (44.8%), JavaScript (44.8%), PHP (20.7%), others (17.2%), and Ruby (13.8%).

We asked students to answer comparison questions about their behaviors on building secure software before training and after training. These questions include 8 steps. Step 1, go through CWE/SANS top 25 most dangerous software errors or any other error list to find security errors. Step 2, prioritize security errors to decide which one to remediate first. Step 3, reading technique details of security errors to understand errors. Step 4, consider security issues when in processes of coding functionalities. Step 5, source code walkthrough after coding process of functionalities. Step 6, use static analysis tools to detect security errors, Step 7, architecture/design review for security issues. Step 8, use dynamic analysis tools to detect security error. Participants could choose from five levels of frequencies for the steps, never as 1, almost never as 2, sometimes as 3, frequently as 4, very often as 5. Figure 8 shows that, before training, participants never or almost never pay attention to the security coding errors. After training, participants more frequently use the 8 steps discussed above (between sometimes and frequently levels). Figure 8 also showed that, after training, participants are willing to perform step 4 and step 5 frequently. Step 1 and step 8 frequencies increased more than other steps.

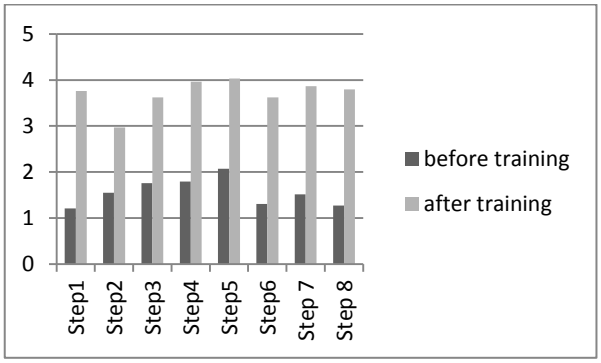


Figure 8 Response result for steps building secure software

We also asked four questions about participants' attitude on the guide and case study. They were asked to choose from five degrees of agreement (1 to 5 for strong disagree to strong agree) for four sentences. "I understand the examples provided in secure software development project on PSP website." degree = 4.42). "The step by step method provided in secure software development project on PSP website helped me to target and

```php
$album_id=$_GET['album_id'];

if(strlen($album_id)>10){
$errors[]='Album ID can not have more than 10 characters'
}
if(album_check((int)$album_id)==false){
    $errors[]='Can not find this album.'
}
else{
    $album_id=(int)$album_id;
}
if(!empty($errors)){
    foreach($errors as $error){
        echo $error, '<br />';
    }
}
else{
    $album_data=public_album_data($album_id, 'albumName', 'description');
    echo '<h3>View album : '.$album_id. '</h3>';
    echo '<h3>',$album_data['albumName'],'</h3><p>',$album_data['description'],'</p>';
}
```

a. PHP code

```
get album id by using HTTP GET parameter

If(Length of album_id > 10){
    save error message into Errors array;
    }
If(function album_check() return false){
    save error message into Errors array;
    }
else{
    Typecasting album_id into integer
}
If(Errors array is NOT empty){
    for each error message in Errors array{
        print out the error message on the webpage
    }
}
else{
    call public_album_data function to fetch album information;
    Print out: View album album_id;
    Print out: Album name and album description;
    }
```

b. Pseudo code

Figure 7 Mitigation code example for CWE-79

order the security error." (Average degree = 4.11). And "I like the way the secure software development project introduces CWE/SANS Top 25 most dangerous software errors." (Average degree = 4.34).

In summary, the training increased participants' motivation to perform the eight secure software developing steps. After training, participants are willing to consider security issues when in processes of coding functionalities and source code walkthrough after coding process of functionalities. The training also significantly enhanced participants' frequencies on reading research resources about security errors and using dynamic analysis tools to detect security errors. And participants hold positive attitude on the step by step guide training and case study. We also conducted other survey questions and analysis. We will not discuss them in details, because of the space limit.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we introduced a step by step methodology for programmers and designers to make use of CWE research resource to solve security-related issues. We provide a path for developers to generate the raw error list, prioritize errors by risk evaluations, target security error in source code, test attack, establish mitigation strategy and document the solved errors. We introduced S-value in risk evaluation to help developers prioritize errors in the raw error list. The S-value is calculated from weighted remediation cost, ease of detection, attack frequency, attacker awareness. We trained developers by a web-based application as a case study to find software security errors and to provide mitigations based on the 2011 CWE/SANS Top 25 Most Dangerous Software Errors. In our case study, we presented error code example, attack example, and mitigation code example for CWE-79 which is the error with highest S-value. The step-by-step approach we introduced separate the complicate security errors targeting and mitigation process into small and easy to follow steps. This approach could fill the gap between software security researches and developers' security practice. On the other hand, S-value with flexible weight of properties could be able to help developers balance efforts to mitigate an error with resource (time and budget) limitation.

We are conducting more surveys and experiments to evaluate our approach on two main aspects. First, we investigate the impact of our step-by-step path in helping developers to use existing resources. Second, we investigate the impact of S-value in helping developers and software managers to prioritize errors in a specific application instead of a general ranking. We also educate developers to use this approach on automatic static analysis tools and automatic dynamic analysis tools.

7. ACKNOWLEDGEMENT

This work is supported by the National Science Foundation, under grant 1220026 and grant 1043945.

8. REFERENCES

[1] A. Franklin, B. Sherrill, and B. Ivey, "**IBM X-Force 2012 Trend and Risk Report,**" Available at https://www.ibm.com/ibm/files/I218646H25649F77/Risk_Report.pdf, 2013.

[2] "**CVE Vulnerabilities By Year,**" 2015. Available at http://www.cvedetails.com/browse-by-date.php.

[3] S. M. Christey, J. E. Kenderdine, J. M. Mazella, B. Miles, and R. a Martin, "**CWE report version 2.8.**"

[4] OWASP, "**Software Assurance Maturity Model,**" 2013. Available at https://www.owasp.org/index.php/Category:Software_Assurance_Maturity_Model#tab=OpenSAMM.

[5] "**OWASP Development Guide,**" 2005. Available at https://www.owasp.org/index.php/Projects/OWASP_Development_Guide.

[6] E. S. Simpson, M. Howard, M. Corp, and K. Randolph, "**Fundamental Practices for Secure Software Development A Guide to the Most Effective Secure Development Practices in Use Today,**" Available at http://www.scribd.com/doc/100731858/Writing-the-Secure-Code, 2011.

[7] B. Sullivan, E. Bonver, J. Furlong, and S. Orrin, "Practices for Secure Development of Cloud Applications Table of Contents," **2013 SAFECode & Cloud Security Alliance**, 2013.

[8] M. Zeng and F. Zhu, "**Secure software development,**" *Pervasive Lab UAH*, 2015. Available at http://pervasive.cs.uah.edu/PSP/content/secure-software-development.

[9] "**CWE Common Weakness Enumeration,**" 2014. Available at http://cwe.mitre.org/.

[10] "**2011 CWE/SANS Top 25 Most Dangerous Software Errors,**" 2011. Available at http://cwe.mitre.org/top25/index.html.

[11] "**CWE-798: Use of Hard-coded Credentials,**" 2011. Available at http://cwe.mitre.org/data/definitions/798.html.

[12] OWASP, "**Static Code Analysis,**" 2015. Available at https://www.owasp.org/index.php/Static_Code_Analysis.

[13] "**Coverity Code Advisor,**" 2015. Available at http://www.coverity.com/products/code-advisor/.

[14] M. D. Ernst, "Static and dynamic analysis: Synergy and duality," **WODA 2003: ICSE Workshop on Dynamic Analysis**, pp. 24–27, 2003.

[15] Veracode, "**Veracode Products,**" 2015. Available at http://www.veracode.com/products/.

[16] T. Rains, "**Microsoft's Free Security Tools – Attack Surface Analyzer.**" Available at https://blogs.microsoft.com/cybertrust/2012/08/02/microsofts-free-security-tools-attack-surface-analyzer/.

[17] Sectools, "**Top 125 Network Security Tools,**" 2015. Available at http://sectools.org/tag/web-scanners/.

[18] J. Walden and M. Doyle, "SAVI: Static-Analysis vulnerability indicator," **IEEE Security and Privacy**, Vol. 10, pp. 32–39, 2012.

[19] P. Anderson, "Measuring the value of static-analysis tool deployments," **IEEE Security and Privacy**, Vol. 10, No. June, pp. 40–47, 2012.

[20] "**OWASP Top 10-2013 The Ten Most Critical Web Application Security Risks,**" 2013. Available at https://www.owasp.org/index.php/Top_10_2013-Top_10.

[21] SAFEcode, "**SAFEcode Homepage.**" Available at http://safecode.org/.

[22] Roger S. Pressman, **Risk management**, Seventh Ed. McGraw-Hill, 2010.

[23] CWE, "**CWE-79 Cross-site Scripting,**" 2014. Available at https://cwe.mitre.org/data/definitions/79.html.

[24] "**CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection'),**" 2011. Available at http://cwe.mitre.org/data/definitions/89.html.