Security Patterns As Architectural Solution - Mitigating Cross-Site Scripting Attacks in Web Applications

Priya Anand*, Jungwoo Ryoo†
College of Information Sciences and Technology
Pennsylvania State University
University Park, PA, USA - 16802
Email:*axg36@ist.psu.edu, †jryoo@ist.psu.edu

***Abstract*—Security patterns are solutions for a recurring security issues that can be applied to mitigate security weaknesses in a software system. With an increased number of security patterns, the selection of a precise pattern to mitigate a vulnerability may become a challenging for software developers. When an appropriate pattern is identified as a potential solution by a software professional, applying that pattern and its level of integration is purely dependent on the software experts skill and knowledge. Also, adopting the security pattern at an architectural level may be a time consuming and cumbersome task for software developers. To help the software developers community by making this pattern implementation to be a relatively easy task, we developed a tool named - SPAAS Security Patterns As Architectural Solution. This tool would automate the process of implementing the selected security pattern in the software system at an architectural level. Our tool was developed to assess potential vulnerabilities at an architectural level and possible fixes by adopting the selected security patterns. This tool checks the possibility of security patterns that have been already implemented in the system and accurately reports the results. In this paper, we demonstrate the use of our tool by conducting a case study on an open-source medical software, OpenEMR. Our analysis on OpenEMR software using the SPAAS tool pointed out the vulnerable source codes in the system that have been missed by some generic vulnerability assessment tools. Using our tool, we implemented the input validation pattern as a solution to mitigate cross-site scripting attacks. Using our pattern application tool, SPAAS, we analyzed OpenEMR software that has 121819 lines of codes. Our experiment on OpenEMR software that are vulnerable to XSS attacks took 2.03 seconds, and reported the presence of 341 spots of vulnerable codes from a total of 121819 lines of source code. We used our tool to implement intercepting validator pattern on those 341 lines, and we could successfully implement the patterns in 2.28 seconds at an architectural level. Our modified version of OpenEMR with security patterns implementation is presented to its software architect and it would be merged as a security solution in the repository. Without a deep understanding of security patterns, any software professional can implement the security pattern at an architectural level using our proposed tool, SPAAS.**

***Keywords*—Security, Vulnerability, Cross-Site Scripting, Software Architecture, Security Patterns**

I. Introduction

Software development with a huge reservation for its security features has been growing in the past few decades. Yet, many software developers find the solution to mitigate security attacks to be a major challenge facing the community. Most of the software practitioners admitted that modifying source codes after the development stage in order to add some security features to be a bad strategy[1]. One possible reason for the software developers' opinion on that could be the level of modification required to accommodate new line of source in a fully developed software system. A particular file/module can be easily modified in a system and may not be a challenging task. To ensure the level of integrity of a newly added line of code, we have to make sure that files it is depending, sharing, or the files inheriting from the modified one are not affected as well. When a software system is updated to address certain weaknesses or to mitigate the potential vulnerabilities, modifications have to be done at a code-level without affecting its depending or dependent files. Failure to ensure that dependent files are not affected might lead to some unexpected and not an easy to fix software defects. Also, presence of a vulnerable code in a system rarely comes alone or found only at one particular line of code [2]. When we attempt to fix a source code in order to mitigate a vulnerability, purely based upon the technical nature of the solution, level of vulnerabilities may be mitigated. This kind of modification applied at a particular line of code is considered to be a local solution. A local solution still leaves room for the presence of similar kind of weaknesses in other parts of the system. When a security loop hole is inadvertently introduced by a developer in a particular file/module during the development stage, it is reasonable to presume that similar codes would have been introduced in other files in the same software system. Even a group of software professionals may be involved in the development process and not necessarily a single person has to code the entire system. As long as, there is a security loop hole introduced in the software design, it is a good practice to make sure that all files in the system is free of this security weakness. Hence, a security solution would never get realized for its intended purpose in the system unless it was applied as an architectural solution.

Many researches have been done to emphasize the importance of architectural solution [3],[4],[5]. Though the solution was applied at an architectural level, the technical nature of the solution and the consistency maintained between the subsequent releases of the system also plays a major role in estimating the effectiveness of the solution. A source code may be modified to become less vulnerable to a security attack. Also, the presence of similar kind of vulnerable code may be identified across the system, and same solution can be applied in a uniform fashion. An interesting part of this solution is to gauge the level of technical rigor introduced by the software developer. Software practitioner may find an ad-hoc solution to be the required modification to mitigate a vulnerability, and that ad-hoc solution may be carried out at an architectural level. Other software practitioners involved in the same software development process may be inclined to follow a different solution based on his own skills and knowledge. When security solutions were not followed consistently at an architectural level, possibly numerous solutions may get introduced to the system. Hence, we need a dedicated security module in the software system to serve as a repository for security solutions implemented in the system. Very importantly, the proposed security module should be comprised of scientifically proven solution to recurring security issues and applied as an architectural solution to the system. In this paper, we propose security patterns [6] as a solution to address security requirements, and security patterns were considered to be proven solution for recurring security issues. Any modifications required to improve the security readiness of the system can be added to this proposed module that contains security patterns, so that it can serve as a reference in the subsequent software releases.

II. Contributions

Despite the advents and significant developments achieved in the field of software engineering, covering the security aspects of an application still remains to be a challenge for software developers. Security principles were recognized as an important quality factor to be included in the software design to strengthen the system against security attacks. To aid the software developers community to implement security models in the system, many tools have been proposed [7],[8]. Security pattern researchers have done significant level of work to facilitate the use of security patterns in the software system [9]. Currently, hundreds of security patterns are available and many of them are described in a text format at an abstract level [6]. One of the challenges in adopting a security pattern is the contextualization of security pattern in the system. With the current version of text-only description of security patterns, instantiating those patterns are left to the software developer. Hence, converting or coding a security pattern from an abstract level into the required programming language is reserved as an additional work for the software developer. Also, software developer has to make some crucial decisions on vulnerable source code that needs a security pattern. A software security practitioner can predict the vulnerable source code, and the security features that can be added by implementing a security pattern. This paper is focused on those difficulties faced by software practitioners in implementing the security patterns. We have developed a new tool named, SPAAS, which is designed to automate the process of adopting security patterns in the underlying software. We designed the tool with an intention to make the security pattern implementation as a one-step modification on a software under development and also on software that has completed its SDLC (Software Development Life Cycle) phase. Our tool serves as a scanner for a selected type of vulnerability and reports the vulnerable spots in the system. Though we provide options in our tool to carry out an architectural adoption of security pattern, user can make a decision on the required files or modules to get updated with security patterns. Our tool acts as a local vulnerability scanner specific to one programming language, and also as a security pattern implementer at an architectural level for the software system.

III. Related Research

Making an application completely secure is practically impossible, but more features can be added to strengthen the software so that it becomes less vulnerable to security attacks. Among various scientific solutions available for security implementation in a system, security patterns were proposed by pattern researchers as an efficient way to make a system more secure [10]. Security patterns represents a logic, certain security behavior, specific intent that can be implemented in the system to satisfy the required security properties [11]. Pattern researchers are developing and releasing more number of patterns to meet the increasing security needs of a system. As more number of security patterns are identified by pattern researchers, the pattern base, which is a collection of security patterns is constantly increasing. Though the pattern base is increasing its size, many patterns are available with only a textual description at an abstract level. In the current scenario, the responsibility of selecting an appropriate pattern lies on software developers skill and knowledge on security. Even for an expert in software development, selecting a pattern from a database is a time consuming process. To help the developers and designers in the process of selecting the patterns, many pattern classifications were proposed [6],[12],[13].

Hafiz et. al. [6] proposed a classification methodology based on CIA and STRIDE [17] model and explained the use of following a model to select a pattern. While this classification seems to be effective, all available patterns should be mapped with at least one attribute in the model. If a pattern is not mapped to any of the attributes, a designer cannot completely realize the use of this model. Regainia et. al [14] developed a seven-step methodology to categorize the patterns based on their sub-properties that are associated through organization of security principles. Using this methodology, a security pattern can be selected to mitigate a vulnerability with respect to the security principles that have to be addressed. In order to facilitate the wide-adoption of security patterns, Cheng et. al [15] proposed a design pattern template to highlight the security-specific information of patterns. The design pattern template helped the developer to check if a specific design and implementation of the pattern is aligned with the intended security principle. Anand et. al [12] proposed a classification of security patterns using 96 unique patterns and developed a security pattern catalog based on 12 family of vulnerabilities. Once the vulnerability is identified by the software practitioner, the relevant pattern can be selected from the proposed catalog.

Researchers on security patterns classification [6],[12],[14],[15],[19] would definitely help the software developers to select the pattern by guiding through the security requirements or the vulnerability to be addressed. All these previous work helps the software designers to select a security pattern. It is contingent upon the condition if the developers are aware of the security principles to navigate through these methodologies to identify an appropriate pattern. Again, the challenge of instantiating the patterns is purely dependent on the software developers choice of implementation. We perceive the need for a research to develop a methodology that exemplifies the security pattern implementation in the system.

When a pattern is recommended to mitigate the vulnerability in a software, a software developer can deploy a pattern to fix the code at a particular spot. If the same code or code of similar nature is present in other parts of the software, the vulnerability still persists in the system. Researches on architectural relation of a buggy code and its impact on the security, emphasize the need for an architectural solution [3],[9]. Xioa et. al [8] developed a tool to find the architectural roots of a buggy code, so that the bug fixed at the root would reflect its impact throughout the architecture. Ryoo et. al [7] conducted a research on the need for an architectural analysis on a system to address its security requirements. Their research reinforces the need for implementing security patterns at an architectural level and the benefits of adopting security patterns. In this research, we developed a tool SPAAS Security Patterns As Architectural Solution, to implement a security pattern in an architectural mode to mitigate an identified vulnerability in the system. Our tool would serve the software developers and designers to successfully implement the patterns that was selected from the security patterns classifications [6],[14] or catalog[12].

IV. Security Patterns As Architectural Solution

Security patterns are provided to help the software architects and designers to integrate the security features in the system. Security patterns researchers are developing more patterns to further strengthen the security pattern repository. As the patterns available is increasing, the selection and implementation of a pattern is still a gap persisting between the patterns researchers and software developers. Many researches have been done to guide the software designers to select a pattern. An appropriate pattern can be selected based on the classification of attacks, type of vulnerability or CWE number. While these researches have rendered many guidance to the developers to overcome the challenges in selecting a pattern, we examined the need to address a security issue. For example, a software system may be vulnerable to a particular CWE issue, and we have to decide on the need for a security pattern to address that issue. A software designer or an architect may view that as an issue that can be addressed by adding few lines of source code. If it is relatively a simple issue, a minor fix based on the developers' skill will suffice. Apart from the fact that security patterns provide a scientific solution, which is basically the benefit of using a pattern, we explore the need to adopt a pattern in the system.

In our previous research we identified a methodology for architectural analysis for security (AAFS)[7]. we divided the methodology into three optional phases, with a minimum of any one phase. They are Tactic-Oriented Architectural Analysis (ToAA), Vulnerability-Oriented Architectural Analysis(VoAA) and Pattern-Oriented Architectural Analysis(PoAA). User can select any one or two or all three phases of this methodology to derive a conclusion on the security requirements at an architectural level. We conducted a case study to realize the effectiveness of our proposed AAFS methodology. In our case study, we applied all three phases on a open-source medical records software named OpenEMR[16]. Research results showed that the software is vulnerable to XSS attacks and it was on the top of the vulnerability list. Our analysis using AAFS methodology indicated that OpenEMR system has vulnerable codes on not only one file or module, but through out the architecture. Instead of following different solutions for all possible vulnerable spot that are prone to XSS attacks, we proposed to adopt intercepting validator pattern in the design. Based on the documentation, intercepting validator security patterns can be explained as a pattern comprising but not limited to following properties, 1 - it is a validation logic used to address every data-type used in the application, 2 - simple mechanism to filter malicious input data irrespective of data type. The graphical representation of intercepting validator pattern is depicted in Figure 1.

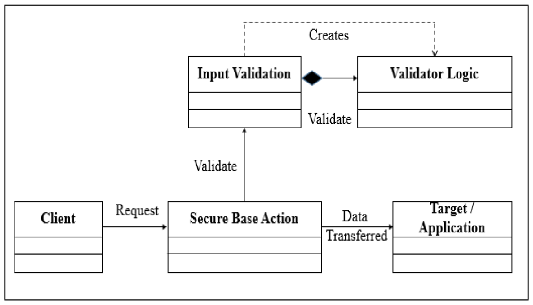

Fig. 1. Intercepting Validator - Class Diagram

We used these details from intercepting validator documentation, and instantiated the pattern in PHP language. We would like to highlight the fact that the research on how accurately the security pattern is coded as per the documentation is beyond the scope of this paper. We have reserved that study in our future research directions. In this paper, we coded a method that comprises the technical description provided in the pattern.

In this research, we focus on how to integrate the instantiated security pattern at an architectural level. To achieve this goal, we used the research results on analyzing the OpenEMR project using AAFS methodology to identify the list of patterns to be adopted. In order to implement a pattern from the list, we developed a tool named SPAAS, that would integrate the patterns at an architectural level.

V. SPAAS Tool

While many tools are available to conduct a security assessment of a software, we designed an additional tool that can serve as a local vulnerability assessment and architectural security pattern implementer for the software. We developed the SPAAS tool using Python 3.6.0. SPAAS is a tool exclusively designed for PHP based web applications. We agree that other vulnerability scanners like IBM AppScan, Nessus and Qualys definitely hold a very high standard in identifying a wide spectrum of bugs, weaknesses and vulnerabilities, and we still emphasize the need for an additional vulnerability scanning function. SPAAS tool was primarily developed for web applications developed using PHP programming language. Hence, we expect our tool to identify the vulnerable spots more accurately.

Algorithm 1 XSS Vulnerability Search Procedure

```
1: function FindVulnerableSpot($File, Directory$)
2:     numberOfLinesVulnerable = 0
3:     if $Input\,file\,is\,a\,directory$ then
4:         for all files in Directory do
5:             MissingSecPattern(file-n1,filepath)
6:     else
7:         MissingSecPattern(InputFile)
8: function MissingSecPattern($file, filepath$)
9:     if $file - n1isadirectory$ then
10:        for files in file-n1 do
11:            MissingSecPattern(file-n2,filepath)
12:            // recursive function call to reach a file
13:        //once a file is reached the following template would be executed
14:        Open the file-n2 in reading mode:
15:        Read all the lines in file-n2:
16:        for each line in file-n2:
17:        if $direct\,input\,from\,user/source$ then
18:            if $securitypatternimplemented$ then
19:                continue;
20:            if $inputfilteringisdone$ then
21:                continue;
22:            if $nopatternorvalidationdone$ then
23:                Store as vulnerable code in output
24:                increment numberOfVulnerables
25:                close the file-n2
26:        return // end of recursive function
27:
```

Our tool has a narrowed scope when compared other leading vulnerability scanners, but it is designed to thoroughly check for a particular vulnerability in the software system. Also, the primary purpose of our tool is not meant to merely identify the vulnerabilities, but to apply security patterns at an architectural level. All available vulnerability scanners conduct security assessment and generates a report. Then the challenge is reserved for the software developer to interpret the report and find a feasible solution. Our tool is aimed to aid the software developer by providing security patterns as a solution. Using our tool, any file can be updated with security patterns in a single file conversion process. Though we highly recommend an architectural implementation of security pattern, we also provide the option to select a particular file or a module to implement security patterns.

Our tool has two major sub-divisions, and they are 1) Identifying vulnerable spots for a selected vulnerability in the software, 2) Applying a selected security pattern. Our current version of the tool checks for Intercepting Validator pattern by default. In order to identify the source codes that are vulnerable to Input Validation, we have to select an input file/module and store the path in the procedure explained in Algorithm 1. On executing this python file, we can get the information of source codes that need intercepting validator pattern. Basically, the tool checks for the spots that accepts input directly from the user or through other sources. If an input is stored in a variable or constant without any validation, it is further evaluated by comparing with the functions represented in the security pattern. As a results, it displays the source codes that are vulnerable to cross-site scripting attacks along with total number of vulnerable codes. Output can be stored in a file, and we can see the results in the command prompt as well. These results can be compared with generic vulnerability scanning tools to verify that we have figured out all vulnerable spots listed in those tools. This process of scanning the vulnerable spots in the system is not a requirement to implement security patterns. We provide this option in the tool to help the user to gauge the need for security patterns by pointing out the vulnerable spots.

Algorithm 2 XSS Vulnerability Search Procedure

```
1: function updateWithPatterns($FilePath$)
2:     counter = 0 // to keep track of updated lines
3:     open the input file in write mode
4:     read all lines from the input file
5:     include the security patterns library file
6:     for each line in the file do
7:         if $linegetsuserinput$ then
8:             if $nosecuritypatternsfound$ then
9:                 if $inputfilteringmissing$ then
10:                    Snip the input part
11:                    apply the security pattern
12:                    as defined in library file
13:                    update the line with security pattern
14:                    increment the counter
15:        Write the line in the file
16:        return
17:
```

To implement the Intercepting Validator Pattern in a source code, we use the procedure defined in Algorithm 2. The file/module/architecture that needs to get updated with security pattern is given as the input path for this procedure. An input can be file or a directory with a combination of sub-directory or files. Our tool is designed to implement the security pattern in a file. If the input is a directory, it opens up all sub-directories, checks for all files in those sub-directories that needs a security pattern and applies the Intercepting Validator pattern in those files. On applying this security pattern, source codes that accepts inputs would be further shielded by the intercepting validator pattern. Using this procedure, we are adding the function representing the security pattern to the input. Our tool basically recreates the source codes by adding security patterns to it. Hence, the function that represents the security pattern should be defined in the library files. We instantiated the security patterns and created a new library file. While updating the code to implement the security pattern, the library file would also be added as part of the modified code.

Using SPAAS tool, source code would be updated with security patterns, and these modified codes can be replaced with previous version of the software. In order to execute the code, we have to add the function representing the security pattern in the library file. As an example, we provide the procedure to instantiate the Intercepting Validator Pattern as depicted in Fig. 1.

VI. Case Study

A. *OpenEMR: Open Electronic Medical Records Project:*

We worked on OpenEMR project to evaluate the effectiveness of our proposed methodology using SPASS tool. OpenEMR is a open source software project, a web application that is primarily designed to meet the data management needs in medical fields. OpenEMR is used to maintain electronics medical records with features including patient appointment scheduling, patient demographics, drug prescriptions details, billing information, and so on. First version of OpenEMR was released in 2001, and followed by 13 updated versions [16]. Current stable version of this software is OpenEMR.5.0. Being a medical records software, ensuring the incorporation of strong security protocols become crucial and necessary.

B. *Results from previous work on OpenEMR:*

We analyzed this project for our previous research work on AAFS methodology[7]. Through our previous research, we derived following results about this project: We identified some vulnerabilities and mapped them to the assessment from vulnerability scanner reports. Our interview with lead architect of this project also validated our results. We mapped those weaknesses to the security patterns and associated 43 vulnerabilities with the intercepting validator pattern. We evaluated the countermeasures adopted in the source code to address these vulnerabilities. From our evaluation, we concluded that parameterization and escaping mechanism were used as framework-based countermeasures. We also evaluated for CWE-87 (XSS)[18]. Our code review and interview with the lead architect revealed that OpenEMR code base has no systematic action against XSS vulnerabilities. Some XSS vulnerabilities have been patched in the past. Specifically, we could not identify a secure coding standard or a systematic code inspection to address XSS vulnerabilities. We shared our assessment done on OpenEMR using AAFS methodology to its lead architect. We emphasized one of our research results that OpenEMR project would continue to experience more XSS vulnerabilities until a radical and architectural solution is introduced.

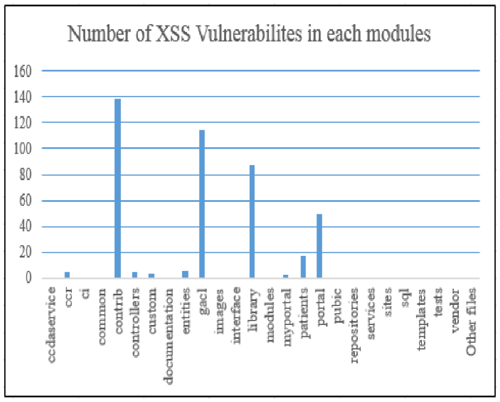

Fig. 2: Number of XSS vulnerabilties identified in each module of OpenEMR

C. *Applying SPAAS on OpenEMR:*

Our results on OpenEMR projects security readiness case study using AAFS methodology was further reinforced as more XSS vulnerabilities were reported in its GitHub repository. Among those reported security bugs in the GitHub, we selected a particular issue that raised the XSS vulnerabilities in a file. SPASS tool was used to address XSS vulnerability reported in the file object_search.php. Though two vulnerable codes were reported in the issue list of OpenEMR GitHub page, SPAAS identified four vulnerable spots in the file. We manually verified the reported code to be vulnerable to XSS attacks. We planned to conduct a thorough analysis at an architectural level. We introduced a new file named securityPatterns.php. In this file, we instantiated the intercepting validator pattern in PHP programming language and turned this file (securityPatterns.php) into a library file. We have covered all required aspects of intercepting validator pattern in this library file. On top of that, if a developer or designer intends to add more validation, it can be done by adding the functions to this library file.

D. *Results:*

In this project, we started evaluating the presence of XSS vulnerabilities in a particular file as reported in the project's GitHub repository. We could both manually and also using SPAAS tool verify the proof-of-concept as it is stated in the issue list. To mitigate the XSS scripting vulnerability, we used Intercepting Validator Pattern in two different spots in the code. As per our hypotheses, we predicted that similar weakness in code in other modules of the project. We used the tool to verify the presence of other XSS vulnerabilities, and the total number of XSS vulnerabilities in the project turned out to be 341. As XSS vulnerabilities are achieved when direct unfiltered input from the user or a stored XSS script is used in the web applications. Instead of working on only one module, or every modules in the project, we decided to focus on the modules that have high number of vulnerabilities in it. Results show only 9 out of 25 modules comprise potential XSS vulnerabilities, with highest number of XSS vulnerable codes from 'contrib' module, followed by 115 vulnerabilities in 'gacl' module. Our results on identified XSS vulnerabilities on individual modules are reported on Fig. 2.

Our results show that security vulnerabilities like XSS got a high potential to be present in the unfiltered source, or the codes without any security patterns. In order to identify these vulnerabilities, we did not evaluate the inter-dependency of modules, or the inheritance of files among the modules. We applied our hypothesis that irrespective of the file dependency, presence of a vulnerable code leaves room for security attacks. Hence, pointing out every possible locations of vulnerable spots and addressing it with an appropriate security pattern becomes essential to mitigate security attacks. By adopting AAFS methodology for an architectural analysis, and applying security patterns using SPAAS tool as architectural solution, improved the security readiness of a software system. In this case-study, we adopted the methodology to address the prevalent security issues of OpenEMR project(XSS vulnerability), and used SPAAS tool to implement Intercepting Validator security pattern.

VII. Limitations

We developed this new approach of deploying a tool for identifying more specific vulnerabilities found in a particular programming language used in web application. We demonstrated the use of this tool while using along with other widely adopted vulnerability scanners like IBM AppScan, Nessus and Qualys. While the vulnerability scanners are meant to provide risk assessment for all kinds of software, our tool, in its current version is limited to assessing vulnerabilities in a particular programming language. We are aware of the fact that many vulnerability scanners are widely adopted by security professionals and those scanners render in-depth insights about the security loopholes in a system. Our primary goal is to add an additional tool that is compatible to the software, as a local analyzer, and also identify the possibility of a particular vulnerability at an architectural level. Our tool serves as a security pattern implementer more than a vulnerability scanner. Our tool in its current version is capable of implementing a particular pattern (Intercepting Validator Pattern as stated in our case study) at an architectural level for PHP based web applications. Our tool cannot support implementing more security patterns and also not compatible for other commonly used web application development languages like Java, JavaScript, Perl etc. We have reserved those implementations for our future work as we continue our research on further developing this tool. Through our case study on OpenEMR software, we could verify the use of our tool as a local vulnerability scanner and security pattern implementation at an architectural level only for PHP programming language.

VIII. Future Directions

As we evaluated the advantages of using a local scanner and pattern implementer in this research, we had laid a framework to further develop our tool by instantiating more security patterns, and also accommodating more web application development languages. Our future research plans include a security enhancement technique by applying our AAFS methodology combined with SPAAS tool in a software system by following a six-step process: 1) Using vulnerability assessment tools, conduct a security assessment of a software system, 2) Perform an architectural analysis of security for a software system using AAFS methodology, 3) identify the required security tactics to be followed and the pattern to realize those tactics, 4) Using SPAAS tool, evaluate potential vulnerable codes that are lacking security patterns, 5) using SPAAS tool, implement the patterns in the software, 6) Using the vulnerability assessment tool used in step-1, conduct the security assessment on the updated software version that comprises security patterns. By following this six-step process, we are targeting to get a catalog of security tactics to security patterns mapping as an intermediate result. In this research, we continued our previous work on OpenEMR[16], that resulted the set of patterns to be applied on the system. Our newly developed SPAAS tool reported the spots that need an input validation pattern, and also applied those patterns. First three steps in our aforementioned six-step methodology were conducted in our previous research [7], and we completed the remaining three steps in this research. In our future research, we are planning to implement our proposed six-step security enhancement technique in more open source systems and validate the results by assessing the security features using vulnerability assessment tools.

IX. Conclusion

This paper presents a tool-based approach to help software designers and developers to implement security patterns at an architectural level. To accomplish this task, the methodology followed was divided into two major parts, 1) architectural analysis for security (AAFS), and 2) using SPAAS tool to implement the security patterns recommended through AAFS analysis. We conducted AAFS analysis on OpenEMR project in our previous research and continued with implementing the recommended patterns in this research. While AAFS methodology identifies the list of vulnerabilities to be addressed based on the risk level, our proposed SPAAS automated the process of implementing those patterns in the software repository. We conducted a case-study on OpenEMR project to realize the effectiveness of our proposed methodology. SPAAS tool identified 341 spots from 121819 lines of codes, that has potential vulnerability to XSS attacks in 2.28 seconds. We perceived the difference in number of vulnerabilities is more with SPAAS tool, as it is exclusively designed for a particular programming language and targeted to identify a specific vulnerability. More than the advantage of an improved scan coverage, this tool helped to implement a security pattern in 341 different spots in 2.28 seconds. By implementing the pattern using SPAAS tool, we used a security patterns as an architectural solution to mitigate XSS vulnerabilities at an architectural level. In the near future, we intend to automate the AAFS methodology and make the entire analysis, AAFS followed by SPAAS, as a fully-automated process to implement security features at an architectural level.

References

[1] Mockus, Audris, Roy T. Fielding, and James Herbsleb. ”A case study of open source software development: the Apache server.” *I*n Software Engineering, 2000. Proceedings of the 2000 International Conference on, pp. 263-272. IEEE, 2000.

[2] Mo, Ran, Yuanfang Cai, Rick Kazman, and Lu Xiao. ”Hotspot patterns: The formal definition and automatic detection of architecture smells.” *I*n Software Architecture (WICSA), 2015 12th Working IEEE/IFIP Conference on, pp. 51-60. IEEE, 2015.

[3] Jansen, Anton, and Jan Bosch. ”Software architecture as a set of architectural design decisions.” *I*n Software Architecture, 2005. WICSA 2005. 5th Working IEEE/IFIP Conference on, pp. 109-120. IEEE, 2005.

[4] L. Bass, P. Clements, R. Kazman, Software Architecture in Practice, Reading,*M*A:Addison-Wesley, 2003.

[5] Brown, Nanette, Yuanfang Cai, Yuepu Guo, Rick Kazman, Miryung Kim, Philippe Kruchten, Erin Lim et al. ”Managing technical debt in software-reliant systems.” In *P*roceedings of the FSE/SDP workshop on Future of software engineering research, pp. 47-52. ACM, 2010.

[6] Hafiz, Munawar, Paul Adamczyk, and Ralph E. Johnson. ”Organizing security patterns.” *I*EEE software 24, no. 4 (2007).

[7] Ryoo, Jungwoo, Rick Kazman, and Priya Anand. ”Architectural Analysis for Security.” *I*EEE Security Privacy 13, no. 6 (2015): 52-59.

[8] Xiao, Lu, Yuanfang Cai, and Rick Kazman. ”Titan: A toolset that connects software architecture with quality analysis.” In *P*roceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 763-766. ACM, 2014.

[9] Xiao, Lu, Yuanfang Cai, and Rick Kazman. ”Design rule spaces: A new form of architecture insight.” In *P*roceedings of the 36th International Conference on Software Engineering, pp. 967-977. ACM, 2014.

[10] Heyman, Thomas, Koen Yskout, Riccardo Scandariato, and Wouter Joosen. ”An analysis of the security patterns landscape.” In *S*oftware Engineering for Secure Systems, 2007. SESS’07: ICSE Workshops 2007. Third International Workshop on, pp. 3-3. IEEE, 2007.

[11] Heyman, Thomas, Riccardo Scandariato, Christophe Huygens, and Wouter Joosen. ”Using security patterns to combine security metrics.” In *A*vailability, Reliability and Security, 2008. ARES 08. Third International Conference on, pp. 1156-1163. IEEE, 2008.

[12] Anand, Priya, Jungwoo Ryoo, and Rick Kazman. ”Vulnerability-Based Security Pattern Categorization in Search of Missing Patterns.” In *A*vailability, Reliability and Security (ARES), 2014 Ninth International Conference on, pp. 476-483. IEEE, 2014.

[13] Washizaki, Hironori, Eduardo B. Fernandez, Katsuhisa Maruyama, Atsuto Kubo, and Nobukazu Yoshioka. ”Improving the classification of security patterns.” In *D*atabase and Expert Systems Application, 2009. DEXA’09. 20th International Workshop on, pp. 165-170. IEEE, 2009.

[14] Regainia, Loukmen, Sbastien Salva, and Cdric Bouhours. ”A classification methodology for security patterns to help fix software weaknesses.” http://confiance-numerique.clermont-universite.fr/index-en.html

[15] Cheng, Betty HC, Sascha Konrad, Laura A. Campbell, and Ronald Wassermann. ”Using security patterns to model and analyze security requirements.” In *I*EEE workshop on requirements for high assurance systems, pp. 13-22. 2003.

[16] http://www.open-emr.org/wiki/index.php/OpenEMR_Wiki

[17] M. Whitman and H. Mattord, *Principles of information security*. Cengage Learning, 2011.

[18] https://cwe.mitre.org/data/definitions/87.html

[19] Anand, Priya, Jungwoo Ryoo, and Hyoungshick Kim. ”Addressing Security Challenges in Cloud ComputingA Pattern-Based Approach.” *S*oftware Security and Assurance (ICSSA), International Conference on. IEEE, 2015.