Proceedings of the 14th Colloquium for Information Systems Security Education
Baltimore Marriott Inner Harbor
Baltimore, Maryland June 7 - 9, 2010

The Impact of a Structured Application Development Framework on Web Application Security

Heather Richter Lipford, Jing Xie, Will Stranathan, Daniel Oakley, Bei-Tseng Chu
{heather.lipford, jxie2, daoakley, billchu}@uncc.edu, will@thestranathans.com

Abstract – Many security vulnerabilities are caused through flaws in the developed software. We investigate the hypothesis that using a structured software development framework reduces the flaws introduced by programmers, leading to more secure software. To test this hypothesis, we conducted an empirical study comparing applications developed using Struts I, a widely used framework for Java-based web applications, against applications written in JSP/Servlet. Our results suggest that a structured framework may reduce security vulnerability density, mainly as a result of using libraries that abstract away low level API calls. Modular design, e.g. the MVC model, had only a modest impact. We also describe the distinct patterns of programming behavior we observed and the implications for security and education. These results provide guidance to the development of programming frameworks and paradigms to promote secure software development.

Index terms – security software development, secure programming.

I. Introduction

Software flaws are at the root cause of many of today's information security vulnerabilities. Seventy-five percent of the SANS 2007 Top 20 Security Risks [1] can be mitigated by code patches. Web applications, because of their wide use, are particularly vulnerable to well known and easily implemented attacks such as cross site scripting. Thus, improving application security can mitigate the threats of many types of attacks. A common intuition among industry practitioners and software developers is that better designed applications will also be more secure. Modular design, such as utilizing the Model-View-Controller (MVC) model, may encourage developers to spend more time on design and less on lower level programming activities, and make debugging easier. Structured development frameworks encourage such modular design. In addition, libraries and other facilities provided by such frameworks may prevent certain kinds of errors and also aid in security. However to the best of authors' knowledge, no empirical research has been done to examine the impact of such factors on security.

Our hypothesis is that utilizing a structured development framework leads to more secure software. To explore this hypothesis, we investigate whether using Struts I, a widely used web application development framework for Java-based web applications [2] leads to fewer software vulnerabilities compared to using the less structured paradigm of JSP/Servlets. We designed an empirical user study to examine the vulnerabilities that intermediate programmers introduce in creating a web application using either framework. If we find support for our hypothesis, then promoting such structured frameworks in education and industry would also have a beneficial impact on security. Even so, it will still be important to understand the precise reasons and impact on security so that we can properly train developers how to avoid common and easily preventable security flaws in their applications. Insights from our study may also benefit designers of future frameworks and programming support environments to integrate best practices into such tools to promote secure programming.

II. Background

Secure software development involves all stages of the application lifecycle [3]. A number of research projects have investigated how to incorporate security into requirements and design methodologies [4, 5], how to design more secure programming languages [6, 7], and how to detect and test for security vulnerabilities [8, 9]. Yet, very few have focused on the task of programming itself, and what techniques will best help developers avoid security problems in the first place. Thus, we are exploring secure programming practices to find the impact of various practices on security. For example, different languages have been shown to have different bug rates [10]. Thus, different languages or development environments may also impact the rate of security vulnerabilities that developers introduce.

Modularity has been shown as a way to deal with the complexity of designs, separating and solving the problems of different concerns [11]. Better understanding

Department of Software and Information Systems
Laboratory of Information Integration Security and Privacy
University of North Carolina at Charlotte

of the program, and more focused efforts on the different concerns presumably leads to fewer bugs [12]. For example, several researchers have explored the use of Aspect Oriented Programming in separating security concerns out from the rest of the application [12, 13]. However, we are unaware of any investigations on the impact of widespread development frameworks or environments on the security of applications. While structured frameworks do provide more modularity than traditional programming languages, security concerns may or may not be modularized, and thus, impacted. In this paper, we focus on the Struts I web application development framework.

A. Struts

Struts I (referred to as Struts hereafter) was designed to build production quality Java-based web applications. The following quote is taken from the web site of the open source Struts project:

> *The project is called "Struts" because the framework is meant to furnish the "invisible underpinnings" that support professional application development. Struts provides the glue that joins the various elements of the standard Java platform into a coherent whole* [2].

Struts utilizes request and response mapping to help designers separate the business logic of their application from the page design elements and navigational logic. Overall it helps designers adhere to the Model-View-Controller (MVC) architecture. It is clear from the history of Struts development that secure software design was not one of the original design goals. Thus, while the framework is modularized, that modularization is not based around security concerns. Yet, this modularization may still lead programmers to focus on particular aspects of the application, introducing fewer vulnerabilities. Additionally, Struts provides a more extensive set of libraries and utilities that may provide more secure functionality that the developer can easily rely upon. Thus, we are interested in identifying features within Struts that promote secure programming through this empirical study.

B. Vulnerability Analysis

To examine our hypothesis, we must also define a suitable measure of software security. Penetration testing and static analysis of source code are two major methods used in industry to find security flaws in software. Static analysis [14, 15] can be carried out by commercial tools which can effectively detect well known classes of security vulnerabilities. Penetration testing is a much more manual process, but can find subtle logic flaws that cannot be reliably discovered by static analysis. For this study we chose to use static analysis because the process is repeatable, it is being widely practiced in industry to measure software security, and static analysis is likely to find nearly all common security vulnerabilities in small web-based applications like the one we use in this study. We chose to use a common industry tool, the Static Code Analyzer made by Fortify [1].

III. EXPERIMENTAL METHODOLOGY

We set out to compare two approaches to building web applications: (a) using Java Servlet and JSP, and (b) using basic features of Struts. Ideally, we would like to understand the flaws introduced in real world, enterprise environments and applications. However, determining the impact of all factors, such as the experience and security training of the programmers, is challenging. Thus, for our first investigation we chose to design an experimental study where we could control as much as possible for other factors, and examine in detail the source of the flaws that programmers introduced. We designed a between-subjects experiment to examine programmers creating the same web application in either of the two environments.

We recruited graduate students from our department with similar levels of programming experience, according to the following criteria: (1) recent CS undergraduate degree, (2) knowledge of building web applications with Java and Servlet, (3) no professional programming experience, and (4) no prior exposure to either Struts or secure coding techniques. We did not seek professional programmers, or those with security experience as we felt those experiences would likely impact the kinds of flaws that students introduced. We further observed during the course of the study that, with one exception, participants' programming abilities were evenly matched. One participant using Struts was unable to complete the project and we dismissed him half way through the study. We do not report his results.

The application we chose to have students create was a micro blog. We created a description of the application, along with a set of the following use cases: sign up for user account, login, logout, change password, post blog, add a friend, follow posts, and view all posts. The application was simple enough to be completed in a limited amount of time, but still contained complex functionality that would have security implications.

We conducted the study in two phases. We first performed a pilot study with four students to ensure the programming tasks were achievable in the time allotted. The experiment went smoothly and we did not have to modify the experiment methodology. We then conducted the study a second time with 8 students. Because there was no substantive difference in the study methods, we combined the data obtained from both studies for

analysis. Because our methodology required a controlled environment and significant amount of time for participants, our participant numbers are still small. However, many such studies of programmers are also forced to utilize small numbers for similar reasons. And we believe we are still able to make both quantitative and qualitative conclusions from this sample.

Since all participants were already familiar with Web application development, we began the study by giving a refresher of JSP/Servlet to bring everyone up to speed. Participants were then split into two groups. Half were randomly assigned to the JSP/Servlet condition and were given no more training. The other half were assigned to the Struts condition, and given an 8-hour tutorial session on essential elements of Struts. The scope of the tutorial was based on commonly available Struts tutorials (e.g. [16]). It was not possible, nor was it our intent, to teach participants all Struts features. Instead we focused on teaching participants how to map requests to actions, and request parameters to values in those actions. The training gave the participants an introductory understanding of how to separate a web application into components according to Models, Views, and Controllers. We did not cover advanced features such as the validator mechanism, declarative exception handling, and double-post prevention. This decision was made for two reasons. First, these topics are optional in building a Struts application. Second, these features were designed, at least in part, to improve application security. For example, encouraging the use of the validator functionality will encourage people to validate user inputs. We wanted to first focus on the core Struts technology.

Both groups of participants were then given the same requirements (use cases) to develop a micro blog application using the Eclipse IDE with Apache Tomcat as web server and HyperSQL as back-end DBMS. Participants performed their programming tasks in a lab with the other participants in their experimental condition. They were asked not to communicate with each other during the study. A staff member was available to monitor the experiment and answer technical questions at all times. Questions were restricted to requirements and infrastructure (e.g. how to use Eclipse features). Internet access was available during the experiment.

Table 1. Average statistics for completed programs.

	Number of Files	Lines of Code
JSP/Servlet	20.8	1,413
Struts	36.4	1,213

Both groups were given 12 hours to complete the application. Participants were paid for their participation, and were given a bonus for producing working applications. No participants were told that the focus of the experiment was on security. Eleven of the twelve participants successfully completed all the use cases. Table 1 lists average statistics of the resulting applications.

A. Vulnerability analysis

We used Static Code Analyzer made by Fortify [1] (referred hereafter as SCA) to detect security vulnerabilities in the applications. SCA performs data and control flow analysis and uses "rules" to identify possible security vulnerabilities. These rules encapsulate specific knowledge of the application, and its execution environment. SCA uses a numeric score to indicate both SCA's confidence of a discovered vulnerability as well as its severity, with 5 being most severe or confident. If the confidence of the discovered vulnerability is between 3 and 5 and the severity is between 2 and 5, the vulnerability will be classified as a Warning. With a confidence between 4 and 5 and a severity in range of 3 and 5, the vulnerability is classified as Hot. Other findings with lower scales are classified as Info. Since virtually all findings within the Info category do not have security implications, we focus on examining findings in the Hot and Warning categories. All the vulnerability naming and categorization are inherited from the Fortify SCA report.

One of the authors is a technical lead of what Chess and McGraw refer to as a "Software Security Group (SSG)" in the Building Security In Maturity Model [17] at a large commercial enterprise. He has experienced in the practice of using SCA to analyze large software development projects. It is a common industry practice to add customized rules manually for each application under study. This is necessary in order to reduce both false positive and false negative results as each program has unique characteristics. There is no well grounded justification that the default rule set is "optimal". Thus, in addition to using the default rules, we also applied this industry practice of using customized rules uniformly to all the example results. Specifically the following types of rules were added to ensure we did not miss common vulnerabilities:

- SQL Injection Sink rules were added to increase the accuracy of the ranking score of SQL injections. If SCA is able to identify a dynamically-constructed SQL query but is unable to find a taint source, it will create a low-confidence SQL Injection finding.
- Trace untrusted data as they go into a store such as http request or http session attributes.
- Privacy Violation Sources – SCA provides characterization rules that look for sensitive property names like "password", but are not complete. Additional such property names were added.

We also performed expert review of SCA results to eliminate false positives.

IV. RESULTS

The average number of vulnerabilities per program, based on both rule sets for SCA, is shown in Table 2. Overall, applications written in Struts have lower vulnerability density than those written in JSP/Servlet.

Table 2. Average vulnerabilities per program.

	Struts Hot Errors	JSP Hot Errors	Struts Warnings	JSP Warnings
Default Rules	17.4	36.1	39.8	71
Customized Rules	13.4	34	43.8	53.1

We also calculated the *issue-to-sink ratio* which is the number of security vulnerabilities reported per one particular program construct. For Struts the ratio is approximately 2-1 where as for JSP/Servlet it is 1-1. Meaning, for every 100 issues reported in Struts programs, on average only 50 fixes are needed; whereas on average every issue in a JSP/Servlet program requires a fix.

As Table 2 shows, the findings for either rule set are fairly similar. Thus, our continued analysis is based on the results from the customized rules. With the small number of subjects in this study, there are large variances in the data. We chose to use the Wilcoxon non-parametric test to test for differences in vulnerabilities from the two groups, as we make no assumptions on the underlying distribution. Table 3 summarizes the median of vulnerabilities per each category, as well as the Wilcoxon rank test result generated by the latest version of open source statistics software R [18]. The five categories with significant differences are discussed below. Tables 4 and 5 list per participant vulnerability counts in both groups.

Table 3. Median number of vulnerabilities per condition for various categories of security problems.

Vulnerability Category	Struts Median	JSP Median	*p*<=
HOT			
SQL Injection	9	7.5	0.78
Cross-Site Scripting	***0***	***10.5***	***0.014***
Password Management: Hardcoded Password	***0***	***9.5***	***0.012***
System Information Leak	0	0	0.40
Race Condition: Singleton Member Field	0	0	0.22
WARNING			
Unreleased Resource: Database	***14***	***20***	***0.098***
J2EE Misconfiguration: Excessive Session Timeout	***1***	***0***	***0.002***
J2EE Misconfiguration: Missing Error Handling	1	1	N/A
Trust Boundary Violation	9	2	0.17
Log Forging	0	0	0.14
Privacy Violation (output to file)	***12***	***0***	***0.048***
Missing Check against Null	0	0	0.92
Poor Error Handling: Throw Inside Finally	0	0	0.36
System Information Leak	1	9.5	0.63
J2EE Bad Practices: Non-Serializable Object Stored in Session	2	0	0.17

A. Cross-Site Scripting

Struts programmers make far fewer Cross-Site Scripting (XSS) vulnerabilities. Only one of the five Struts participants had XSS errors, whereas XSS vulnerabilities were committed by every JSP/Servlet programmer. Struts encourages the use of its tag library instead of low level java functions to output html streams. The tag library has built-in encoding functions to disable potential XSS attacks. However, a Struts programmer can still use the underlying Java functions if desired. During our Struts introduction we gave a "hello world" example using the tag library, as is typically done for introductory Struts training. Most participants in the Struts group wrote their programs based on this pattern. However, there was one exception where a Struts participant chose to use the Java functions instead, leading to XSS vulnerabilities. This suggests that the security benefits of using the tag library should be emphasized when providing introductory Struts training. We did not find such references in popular Struts training materials (e.g. [16, 19]).

B. Database issues

The use of the database resulted in two common vulnerabilities: hard coded passwords to access the database, and failure to release database handles. All Servlet/JSP participants hard coded the password to the database, whereas only 2 of the five Struts participants had hard-coded passwords. Close examination of the source code revealed that this is due to a peculiar default in the database connection API of Struts. One version of the Java database connection API only needs to provide the URL to the database without supplying login credentials. This only works if the database maintains the default login credentials. The three Struts participants who used this version of the Java API then had no hard-coded passwords.

Table 4. Vulnerabilities per participant for Struts.

Vulnerability Category	P1	P2	P3	P4	P5
HOT					
SQL Injection	10	7	9	3	15
Cross-Site Scripting	***0***	***0***	***9***	***0***	***0***
Hardcoded Password	***0***	***0***	***0***	***6***	***8***
System Information Leak	0	0	0	0	0
Race Condition: Singleton Member Field	0	0	0	0	0
WARNING					
Unreleased Resource: Database	***14***	***0***	***0***	***16***	***23***
Excessive Session Timeout	***1***	***1***	***1***	***1***	***1***
Missing Error Handling	1	1	1	1	1
Trust Boundary Violation	9	18	19	1	8
Log Forging	0	8	0	0	13
Privacy Violation (output to File)	***0***	***12***	***13***	***0***	***21***
Missing Check against Null	0	0	1	5	0
Poor Error Handling: Throw Inside Finally	1	0	0	0	0
System Information Leak	1	0	8	4	0
Non-Serializable Object Stored in Session	0	6	6	0	2

Table 5. Vulnerabilities per participant for JSP.

Vulnerability Category	P1	P2	P3	P4	P5	P6
HOT						
SQL Injection	6	12	9	6	10	6
Cross-Site Scripting	***1***	***10***	***21***	***11***	***9***	***12***
Hardcoded Password	***13***	***13***	***7***	***9***	***10***	***9***
System Information Leak	0	0	0	9	0	0
Race Condition: Singleton Member Field	0	0	0	0	19	2
WARNING						
Unreleased Resource: Database	***22***	***18***	***14***	***27***	***29***	***18***
Excessive Session Timeout	***0***	***0***	***0***	***0***	***0***	***0***
Missing Error Handling	1	1	1	1	1	1
Trust Boundary Violation	16	2	5	1	2	1
Log Forging	0	0	0	0	0	0
Privacy Violation (output to File)	***0***	***0***	***0***	***0***	***0***	***0***
Missing Check against Null	14	0	0	0	8	0
Poor Error Handling: Throw Inside Finally	0	0	0	0	0	0
System Information Leak	53	0	0	0	60	19
Non-Serializable Object Stored in Session	0	0	4	0	0	0

All six participants in the Servlet/JSP group did not release database handles after use, exposing the application to denial of service attacks. In contrast, two of the five participants in the Struts group had zero errors in this category. However, we do not believe there was anything in the Struts framework that encouraged these programmers to release database handles. We believe the explanation is that the Struts tutorial included an example of using a finalize block to release database handles. Some participants may have followed that pattern. In other words, we believe the training example influenced this good behavior.

Overall, Struts participants made fewer database connection calls than JSP/Servlet participants due to design modularity, thus reducing the density of errors related to the hard-coded passwords and database handles. In this way, the modularity would have a benefit on security by limiting the locations of errors and also reducing the number of bug fixes needed to correct errors. However, any database errors are still problematic as it only takes one instance of a hard-coded password to compromise a system.

C. Timeouts

For the vulnerability category "J2EE Misconfiguration: Excessive Session Timeout", Table 4 shows that the Struts participants each had this flaw, while none occurred in JSP/Servlet group. In this case, the libraries of Struts negatively impacted the vulnerability. The default Struts session time out is much longer than the default value for JSP/Servlet, and was flagged as problematic by SCA. Not surprisingly, all participants used respective default values. Thus, this vulnerability would be easily remedied by changing the default Struts behavior. This issue demonstrates the need to understand the implications of default behaviors of any development framework, as programmers will often rely on default values.

D. Privacy violations

Struts programs had more privacy violation errors than JSP/Servlet programs, which had none. Struts has an easy to use logging framework. The availability of this framework encouraged Struts participants to use this facility for debugging, leading to the output of potentially sensitive information. JSP/Servlet participants, on the other hand, used the Java print function for debugging, leading instead to larger counts of the "System Information Leak" vulnerability. Our Struts tutorial pointed out the availability of the logging facility, which participants then used, but did not cover the details of logging. In Struts, programmers can specify one of two modes for logging: debugging and regular logging. When a Struts-based application goes to production, debugging logs can be suppressed leaving only regular log entries. All of the Struts participants used regular logging instead of the debugging logging, which would have been better behavior. Thus, tutorials on Struts should ensure that these modes and purposes of logging are clearly conveyed to programmers to reduce the information leaks.

E. Other vulnerabilities

Although there were not statistically significant differences between the two groups for the remaining vulnerabilities, we observed several interesting differences in programming behavior. A larger study may be able to provide further evidence whether these differences are statistically significant.

None of the participants using Struts committed the Singleton Race Condition vulnerability common for Java programmers. Two of the programs written in JSP/Servlet committed this vulnerability. This type of Race Condition occurs when programmers use a global variable shared by multiple threads of Servlet instances. Such a pattern, unfortunately, has been found in introductory text books (e.g. the text book used at our university to teach introductory web programming). Struts participants do not have to use Servlets in creating their applications, although they are not precluded from doing so. Our Struts participants did not use Servlets, thus they were not going to introduce this particular kind of vulnerability.

Struts participants, however, committed more Trust Boundary Violations. SCA regards HTTP request and HTTP session attributes as a trust boundary, meaning data should be trusted before being stored in the session. Due to the MVC modular design structure, Struts participants tended to use the web session as a global variable to share information among different modules of the application, leading to more trust boundary violations. In contrast, JSP/Servlet participants did not have to rely on the session variable for inter-module communication and so did not commit this error. If supported by larger studies, the secure use of session variables could be another important issue worthy of highlighting for training Struts programmers.

Finally, participants in both groups did not perform proper input validation, leading to significant numbers of SQL injection vulnerabilities. As mentioned earlier, we did not train the Struts participants on the validator mechanism, which would have encouraged participants to do input validation and reduced this vulnerability for the Struts group. We also manually examined Cross Site Request Forgery vulnerabilities, which were created in both groups by incorrect secure session management.

V. IMPLICATIONS

One of our goals in this study was to identify whether a structured framework would provide security benefits due to its modularity and additional functionality. While the results are mixed, we believe that our study does provide some evidence that Struts would help to reduce security vulnerabilities over JSP/Servlet, but with several important lessons.

The utilities and libraries provided by Struts appeared to offer the most security benefits. The use of the Struts tag and database libraries protected against certain vulnerabilities such as Cross-Site Scripting. The use of other Struts libraries, such as the validator facility, would have also protected against additional vulnerabilities if programmers would have had additional training. However, those libraries themselves did cause a few problems; namely, the default session timeouts were too long. Thus, for any development framework or programming paradigm, it is important to understand the security implications of the provided libraries and utilities, and set default values that reflect good security practices. Still, our participants were not required to use the libraries of Struts, and did occasionally resort to using Servlets for example. Thus, programmers utilizing such development frameworks should be encouraged to use the added functionality, and trained on the benefits they provide.

The modularity of the Struts framework was less beneficial in our experiment. Our results suggest that design modularity may lead to reduced vulnerability density. In other words, the modularity did lead to a better *issue-to-sink ratio*, meaning that the MVC framework appears to lead to "less bad code" and would make vulnerabilities easier to fix. In programs written using Struts, the identified vulnerabilities were concentrated in a few Data Access Object files. In the JSP/Servlet applications, the vulnerabilities were spread throughout many different JSP files, making them more

difficult to find and correct. In a real enterprise environment, this cycle of searching and finding and debugging vulnerabilities would play a larger role in the final security of the end product. Thus, modularity may not have had as great an impact on this small-scale experiment, where programmers were merely trying to get their application to work and ending there. Additional study needs to be done to determine the impact of such modularity on larger scale production systems.

One key conclusion from our study which we were not expecting is the importance of training on the programming behaviors, and resulting applications. The Struts participants used a provided database pattern that included the code to release the database handles. They also incompletely used the Struts logging facility due to a lack of training on the different logging modes. JSP/Servlet participants may have used their previous knowledge of examples with Singleton errors from their textbooks. In other words, even basic training on languages and environments not aimed at security concerns can have security implications. Sample code will get learned and reused, which can have either positive or negative effects. Thus, our study further demonstrates the need for secure programming practices to be taught not just in training and courses dedicated to security. Instead, security should be incorporated into basic education and training through the use of good examples and practices, even if the students are not entirely aware of the security benefits.

Our study also revealed that security training is still paramount, even in such structured development environments. Studies such as this one are important to identify the security vulnerabilities that are common for programmers to create in different paradigms and environments, so security training can be tailored to the likely problems. Our results identified the importance of training Struts users in the security implications on the tag library, database handles, logging, and session variables. For example, the Struts participants committed Trust Boundary Violations due to their treatment of the session as global variables. Thus, we would recommend that basic training include examples showing the sharing of information across modules that does not commit this vulnerability. Additional research on larger applications needs to be done to identify additional common vulnerabilities.

VI. Methodological Limitations

Our study highlights several issues and challenges in examining research questions regarding secure programming behaviors. The controlled experiment allowed us to control for a variety of factors, such as the education and experience of the participants, in comparing these two programming environments. It also allowed us to more directly observe the behavior of the programmers. Such controlled results are important in comparing the impact of various tools and methods on individual programmers. Our methodology has several limitations however. The application itself was relatively small, and not representative of the greater complexity inherent in enterprise applications. The number of participants was also small, due to the amount of time required by any person to receive the training and perform the study tasks. The cost of paying additional participants, or paying non-student programmers to participate, may be prohibitive. Still, even with these limitations, we were able to observe several statistically significant differences between the groups. We were also able to examine each application, and understand the training each participant received, and provide explanations for the vulnerabilities we found. Thus we believe that such controlled experiments for comparing techniques, languages and environments can provide significant contributions that complement other methods.

VII. Conclusion

Our work suggests that a structured framework may reduce security vulnerability density. This benefit is mainly derived from using libraries that abstracted away lower level API calls. Modular design, e.g. the MVC model, had a modest impact on software security in our experiment, potentially reducing debugging costs of finding and fixing security vulnerabilities. Design modularity and program maintainability have been among the top motivations for creating application frameworks. We suggest security should be added as another key motivation. Toward that goal, application framework designers should concentrate on adding higher level interfaces by providing important secure programming functions such as input validation, output encoding and exception handling.

Either as a side effect or by design, a development methodology promotes certain programming behavior. Understanding such behavior, through empirical studies, is important for designing more effective secure coding training associated with the development methodology. Often the first application introduced in a tutorial has a lasting impact on the mindset of programmers. These “hello world” examples must be designed carefully and steer programmers toward best practices of secure programming. Training materials should be appropriately designed to educate programmers on secure coding principles in conjunction with key framework elements. For example, in the case of Struts, security benefits of using the tag library as well as appropriate usage of the logging facility should be emphasized.

Many relevant issues were outside the scope of this investigation. For example, we did not address the experience factor and the role it may play in either development environment. In our study, the Struts participants were novices to Struts and may have introduced more vulnerabilities because of the lack of experience. In contrast, the JSP/Servlet participants all had previous experience in that environment. Professional Struts programmers may produce more secure code because they are more experienced in using the libraries provided or have broader experience in the modularity of the designs. They may also be more motivated programmers, as they took the time to get the necessary training to be proficient in Struts. Although we were careful about deciding what to cover in the Struts tutorial, in hindsight we may have covered more information than we ought to, for example the mention of the provided logging facility in Struts which resulted in more vulnerabilities. Overall participants were highly influenced by the examples we provided in tutorials. More empirical studies are needed to gain a deeper understanding on how structured frameworks impact code security, and how that in turns impacts the education of secure programming.

VIII. References

[1] Fortify software, http://www.fortify.com

[2] Struts, http://struts.apache.org/1.x/index.html

[3] Apvrille, A., and Pourzandi, M. Secure Software Development by Example. IEEE Security and Privacy, Vol 3, No. 4, pp 10-17. (2005)

[4] MAC and UML for secure software design. Doan, Thuong, Demurjian, Steven, Ting, T.C., Ketterl, Andreas. FMSE'04: Procedings of the 2004 ACM Workshop on Formal Methods in Security Engineering, 2004.

[5] UMLpac: An approach for Integrating security into UML class design. Peterson, Matthew J.; Bowles, John B.; Eastman, Caroline M., proceedings of IEEE SOUTHEASTCOM 2006.

[6] Dynamic Security Labels and Static Information Flow Control, Lantian Zheng, Andrew C. Myers. International Jurnal of Information Security, 6(2-3), March 2007. Springer.

[7] Jif Reference manual, Jif 3.0.0 Version, June 2006. Stephen Chong, Andrew C. Myers, K. Vikram, Lantian Zheng.

[8] MEDS: The Memory Error Detection System. Jason D. Hiser, Clark L. Coleman, Michele Co, and Jack W. Davidson. Proceedings of the 1st International Symposium on Engineering Secure Software and System. 2009.

[9] An XML-based testing strategy for probing security vulnerabilities in the diameter protocol. Daping Wang, Bell Labs Technical Journal; Fall 2007, Vol. 12 Issue 3, p79-93, 15p, 2 charts, 7 diagrams.

[10] Kratkiewicz, K., Lippmann, R.: Using a Diagnostic Corpus of C Programs to Evaluate Buffer Overflow Detection by Static Analysis Tools. In: Workshop on the Evaluation of Software Defect Tools (2005).

[11] D.L. Parnas, On the criteria to be used in decomposing systems into modules, Communications of the ACM, Volume 15, No. 12. (1972).

[12] B. De Win, B. Vanhaute and B. De Decker, How aspect-oriented programming can help to build secure software, Informatica Volume 26, No. 2, pp. 141-149 (2002).

[13] J. Viega, J.T. Bloch and P. Chandri, Applying Aspect-Oriented Programming to Security, Cutter IT Journal, Volume 14, No. 2, pp. 31-39. (2001).

[14] Chess,B., West, J.: Secure Programming with Static Analysis. Addison Wesley, Boston (2007).

[15] Chess, B., McGraw, G.: Static Analysis for Security. IEEE security & privacy. 2(5) pp. 76-79, (2004).

[16] Struts tutorial: http://www.exadel.com/tutorial/struts/

5.2/guess /strutsintro.html

[17] The Building Security In Maturity Model. http://www.bsi-mm.com

[18] The R Project for Statistical Computing, http://www.r-project.org

[19] Cavaness, C.: Programming Jakarta Struts. O'Reilly, Sebastopol (2003)

[20] Walden, J. Messer, A., Kuhl, A.: Measuring the Effect of Code Complexity on Static Analysis. In: Massacci, F., Redwine, S. Jr., Zannone, N. (eds.) Engineering Secure Software and Systems. LNCS 5429, pp. 195-199 (2009).

[21] Kratkiewicz, K., Lippmann, R.: Using a Diagnostic Corpus of C Programs to Evaluate Buffer Overflow Detection by Static Analysis Tools. In: Workshop on the Evaluation of Software Defect Tools (2005).

[22] Wilander, J., Kamkar, M.: A Comparison of Publicly Available Tools for Static Intrusion Prevention. In: Proceedings of the 7th Nordic Workshop on Secure IT Systems, Karlstad, Sweden, pp. 68-84 (2002).

[23] Zitser, M., Lippmann, R., Leek, T.: Testing Static Analysis Tools using Exploitable Buffer Overflows from Open Source Code. SIGSOFT Software Engineering Notes 29(6), 97-106 (2004).