Eliminating SQL Injection and Cross Site Scripting Using Aspect Oriented Programming

Bojan Simic, James Walden
Department of Computer Science
Northern Kentucky University
Highland Heights, KY

Abstract. Security vulnerabilities in the web applications that we use to shop, bank, and socialize online expose us to exploits that cost billions of dollars each year. This paper describes the design and implementation of AspectShield, a system designed to mitigate the most common web application vulnerabilities without requiring costly and potentially dangerous modifications to the source code of vulnerable web applications.

AspectShield uses Aspect Oriented Programming (AOP) techniques to mitigate XSS and SQL Injection vulnerabilities in Java web applications. AOP is a programming paradigm designed to address cross-cutting concerns like logging that affect many modules of a program. AspectShield uses the Fortify Source Code Analyzer to identify vulnerabilities, then generates aspects that weave in code that mitigates Cross-Site Scripting and SQL Injection vulnerabilities. At runtime, the application executes the protective aspect code to mitigate security issues when a block of vulnerable code is executed.

AspectShield was tested with three enterprise scale Java web applications. It successfully mitigated SQL Injection and Cross-Site Scripting vulnerabilities without significantly affecting performance. The use of AspectShield in these enterprise level applications shows that AOP can effectively mitigate the two top vulnerabilities of web applications in a cost and time effective manner.

Keywords: cross site scripting; xss; sql injection; SQLI, application security; aspect oriented programming; AOP; aspectj; java; web application security

1 Introduction

Most web applications contain security vulnerabilities. A recent paper shows that 71% of education, 58% of social networking, and 51% of retail websites are exposed to a serious vulnerability every day [2], and that 64% of websites have at least one information leakage vulnerability [3].

Securing web applications is an important but complicated task for any development team. While new applications can be designed with security in mind, a significant fraction of software consists of legacy applications that were not designed to be secure. This paper describes AspectShield, a system that can be applied to both new and legacy web applications to mitigate some of the most common vulnerabilities without modifying the source code of those applications.

In theory, legacy web applications can be rewritten to be secure. However, vulnerability remediation is expensive, with estimates of the cost of remediating a single security vulnerability ranging from $160 to $11,000 per vulnerability, depending on the type of vulnerability and its interaction with other code [22]. Web application security consultant Jeremy Grossman noted "The struggle is how do you deal with an enormous number of sites riddled with vulnerabilities? You can't just recode them. It's a dollars and cents issue."

Modification of legacy web applications also introduces the risk of altering the behavior of the application and introducing new defects [21]. Many organizations prefer to avoid modifying legacy applications where possible. Development teams are often afraid of modifying legacy applications, which unfortunately exacerbates the problem by reducing experience with the legacy application, sometimes to the point where no one who remains in the organization understands the application's design or code [21].

We designed AspectShield to mitigate vulnerabilities while avoiding the risk of altering application behavior and avoiding the cost of remediating vulnerabilities through alteration of the source code. This protection is implemented using Aspect Oriented Programming (AOP) techniques. The use of AOP allows for security logic to be developed independently of business logic. This separation of concerns produces a code base uncluttered by logging, input validation, access control, and error handling logic. While AspectShield does not modify application source code, it must have access to the source code in order to identify vulnerabilities and to recompile the application while weaving in vulnerability mitigating aspects generated by AspectShield.

We chose to use AspectJ in this paper, as it is a mature implementation of AOP, that has been in development by the Eclipse Foundation for over a decade. AspectJ is the most widely used AOP system for the Java programming language. An AspectJ aspect is composed of two major pieces:

1. The **pointcut** of an aspect is a pointer to well-defined sections of the application's source code (join points). In our system, a well-defined piece of code can be the name of a vulnerable method name with a particular signature.
2. The **advice** of an aspect defines the specific logic that is to be applied at each join point identified by a corresponding pointcut. There are three types of advice called *before*, *after*, and *around* that execute this logic before, after, or instead of the join point. The AspectShield system uses the around advice to execute validation algorithms in place of vulnerable sections of code identified by the Fortify SCA.

Aspects are woven into the byte code of the application at compile time, while the advice logic is executed at runtime at each block of code identified by pointcuts of the aspects.

The remainder of this paper is composed of the following sections. Section 2 will describe how the creation of the vulnerability mitigation aspects is accomplished. Section 3 will describe the design of an AspectShield aspect. Section 4 will go into detail about the algorithms used to mitigate SQL Injection and XSS attacks. Section 5

will provide the validation and results of an AspectShield implementation on several open source projects. Sections 6, 7, and 8 will describe related work, future work, and conclusions, respectively.

2 Generating the Security Aspects

AspectShield consists of three major steps: use of an external static analysis tool, vulnerability location based on the output of static analysis tools, and generation of aspects to mitigate the vulnerabilities and weaving of the aspects into the locations of the vulnerabilities.

Step 1 - Source Code Analysis

We first locate vulnerabilities using the Fortify Source Code Analyzer (SCA) static analysis tools. Fortify SCA is the winner of the 2011 CODiE awards for "Best Security Solution" [32] and identifies more vulnerabilities than any other detection method. The tool scans the web application source code for vulnerabilities, generating an XML report as output. Counts of vulnerabilities of each type found by Fortify SCA for the three open source web applications we used in this study are shown in Table 1 below. We ignored vulnerability reports of other types for this paper, though we plan to study them in future work.

Table 1. Fortify SCA Results

Application Analyzed	XSS Vulnerabilities	SQLI Vulnerabilities
Alfresco ECM	10	12
Apache OfBiz	869	737
JadaSite E-Commerce	11	76

Step 2 – Analyzing the SCA Results

Fortify SCA reports detailed information about vulnerabilities, including category, file location, and line number. When we analyzed the Fortify SCA reports for a number of web applications, we found that the root causes of XSS and SQL Injection vulnerabilities were a small set of functions. Functions identified as root causes include `executeQuery()` for SQL Injection and `request.getParameter()` for XSS. We compiled a list of potentially vulnerable functions, which were stored in XML files that AspectShield uses to generate pointcuts to mitigate the vulnerabilities. The resulting XML files contained nine functions where the static analysis tool found XSS vulnerabilities and eight different definitions that correspond to potential SQL Injection vulnerabilities. For each of the functions, information such as the function name, number of parameters, and parameter

types was recorded. If additional functions are discovered in the future, they can be added to the XML configuration files.

When AspectShield starts, it parses reports of XSS and SQL Injection vulnerabilities from the XML output of Fortify SCA. AspectShield asks the user to select a mitigation type for each vulnerability, which is applied by weaving in an aspect to apply that mitigation using the location information found in the SCA output.

Step 3 – Running the Aspect Generator

For each vulnerability reported by Fortify SCA, the user will be prompted to select a mitigation type. Different types of mitigations are available for XSS and SQL Injection vulnerabilities. AspectShield is designed to be used by a developer with prior experience with the application that was analyzed. As it is possible for users to select an incorrect mitigation, this user should be the person responsible for the security of the application and have training in security coding and best practices. Unfortunately, there is no universal input validation or encoding technique that could be applied to all vulnerabilities, so AspectShield must ask the user for assistance.

For XSS vulnerabilities, an AspectShield user will be provided with a list of 14 options that range from various types of encoding to whitelisting. These options are implemented using the OWASP Enterprise Security Application Programming Interface (ESAPI) library [9]. ESAPI is a free, open source library of security controls that is widely used by organizations ranging from American Express to the World Bank. It is BSD licensed, enabling AspectShield to use it without introducing licensing issues for commercial software. ESAPI features that we use to mitigate XSS include JavaScript, CSS, HTML, and other types of encoding, along with whitelist rulesets for validating data types such as email addresses and alphanumeric data. AspectShield also allows the user to provide a custom regular expression for validating input, since no library can anticipate every data type accepted by web applications.

For SQL Injection vulnerabilities, a user will be provided with the option to encode the SQL query for either the Oracle or MySQL dialects of SQL. This limitation arises from the fact that the ESAPI library only supports these two dialects of SQL. While encoding is the only option available to the user for mitigating SQL Injection, AspectShield implements additional measures to prevent exploitation of SQL Injection vulnerabilities. These measures include the removal of multiple queries, tautology detection, and the removal of SQL comments before a SQL query is executed.

Once the user selects the mitigations to implement for each vulnerability, AspectShield uses its pointcut and advice templates to generate two aspects for XSS and SQL Injection mitigation. Each selected mitigation is written to a map that is defined and populated in the corresponding aspect's constructor. The advice logic identified in the advice template will then reference this map to determine which type of mitigation should be applied based on the location of the join point.

Once the aspects have been generated, the application is ready to be recompiled with AspectJ to weave in the aspects. AspectShield provides a static JAR file containing the mitigation algorithms that will be linked into the application during

recompilation. The implementation of these algorithms is defined in the following section.

3 AspectShield Design

In order to generate the SQL Injection and XSS mitigation aspects, we created templates for the pointcut and corresponding advice for each of the vulnerable methods that are intercepted to mitigate vulnerabilities.

The pointcut template contains placeholders for the method name, the method signature, the pointcut designator, and a within string. All of the pointcuts in AspectShield use the "call" designator, which allows a method to be intercepted whenever it is called. The within string placeholder will be replaced with a list of names of files in which the method should be intercepted. The method name and parameters will be retrieved from AspectShield's XML configuration files describing potentially vulnerable functions. With the pointcut template created, it will be used to create join points for all of the pieces of code where vulnerabilities were reported.

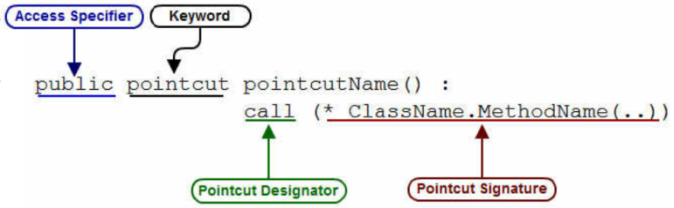

Fig. 1. Example of an AspectJ pointcut.

An aspect's advice will be executed at every join point matched by the pointcut in the application's source code. The around advice used in this implementation executes code in place of the join point it operates over. Since it can have a return value, it must be given a return type (Figure 2).

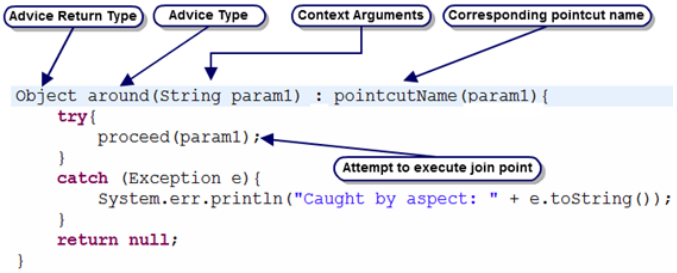

Fig. 2. Example of an AspectJ Advice

Inside of the around advice, the original join point can be executed using the proceed function which takes the same arguments as the join point. Much like the pointcut template defined above, the advice template contains placeholders that are populated when AspectShield is executed. The advice will have a corresponding name equal to the pointcut that it will execute upon. Depending on whether the aspect being

created is for XSS or SQL Injection mitigation, the advice will make a call to the appropriate validation algorithm that will do the mitigation. The algorithm will be provided the original, potentially malicious, parameters for each join point and will return a safe value.

The advice also contains logic to determine whether or not the user elected to provide mitigation for a particular join point. In the event that the call to the mitigation algorithm fails, the advice will execute its `proceed()` method with the pointcut's default parameters in order to maintain the application's normal execution flow. This ensures that AspectShield will not break any of the application's functionality. AspectShield's first priority is to maintain application functionality even in the unlikely event that its mitigation algorithm fails, as security fixes should not break the application. However, if desired, the tool could easily be modified to prevent the code from executing in this scenario. The advice also logs each mitigation using the log4j logger, so a user can detect when a mitigation attempt fails.

4 SQL Injection & XSS Mitigation Results

This section describes the algorithms that are invoked by the XSS and SQL Injection mitigation aspects. Both aspects have similar success criteria for application performance, correct execution of application code, and vulnerability mitigation. When an AspectShield aspect is invoked at runtime, it will receive the potentially malicious input and the mitigation type to be applied as parameters.

4.1 The SQL Injection Mitigation Algorithm

There are three primary choices of mitigation technique for SQL injection vulnerabilities. The first is to use parameterized queries or prepared statements. This method ensures that the attacker is not able to modify the query that is being executed. A second approach is to use stored procedures, where the queries are stored in the database itself and then called by the application when needed. To implement either of these approaches in legacy code, significant work is required. The approach taken in this implementation is to escape all user supplied input before executing any query.

SQL Injection mitigation will be accomplished by the SQL Injection mitigation algorithm when it is invoked by the SQL Injection aspect. The library used to encode all input is the ESAPI encoder library that can do encoding for Oracle or MySQL dialects. The steps of the SQL Injection mitigation algorithm are:

1. The SQL Injection aspect generated in the previous section will call its `doSQLInjectionFix()` method, passing it the query that needs to be validated and the encoding type specified when the user ran AspectShield to generate the aspects.
2. The validator will then test the query for any comments and remove them if found. The query will be passed to the JSQL Parser library that will parse the query and return a list of expressions that the query contains.

3. Each expression in the query will be encoded using either the MySQL or Oracle encoder depending on the choice made by the user when AspectShield was run.
4. Each expression will be tested to determine if it is a tautology, as SQL Injection exploits frequently use tautologies while normal SQL queries do not. This is done by using the Java ScriptEngineManager's Javascript engine to evaluate the expression's value. If the result is always true, the expression is marked as a tautology and removed from the original query.
5. Once all expressions are encoded and tautologies removed, the query is reconstructed using the safe values and returned to the SQL Injection mitigation aspect.
6. When the aspect receives the newly safe version of the query, it will invoke it's proceed() method and pass it the new, safe value.

The mitigation algorithm was timed at each step and performance was evaluated for three case study projects. In the event that the algorithm fails due to an inability to parse the query or for any other reason, it will catch any exceptions, log the failure using a logj4 logger, and return the original query passed in. The original query passed into the algorithm is returned so that if the algorithm fails, the application's normal execution flow will not be affected.

4.2 The XSS Mitigation Algorithm

The difficulty in preventing XSS comes from the fact that such a large number of attack vectors exists. An attacker could potentially steal the session of a victim, manipulate files on the victim's computer, record all keystrokes the victim makes in a web application, or probe a company's intranet where the victim is located [52]. Appropriate validation and encoding can address most reflected and stored XSS vulnerabilities. The algorithms described in this section use the ESAPI encoder and validator libraries to perform escaping and encoding of dangerous data.

The XSS mitigation aspect contains pointcuts that intercept functions, such as `getparameter()` from the request object and `println()` that were identified by the Fortify SCA. At each join point, the aspect's advice logic will implement either encoding or whitelisting on the value intercepted by each pointcut. The process is outlined in the steps below:

1. The advice will call a `doXSSFix()` method for each join point. The method will be passed the intercepted parameter, as well as the type of fix to implement as chosen by the user during the aspect generation phase.
2. Depending on the type of fix the user selected for the join point, the XSS Validator will apply either encoding for the chosen format or validation using a whitelist. The user has the option to choose from several types of encoding or whitelist provided by the ESAPI library [53] [54].
3. If the desired mitigation is a whitelist, the algorithm will check the input against a particular regular expression. If the input fails to match, the code will remove any characters that do not match the desired character set.

4. If the desired mitigation is a particular type of encoding (CSS, JavaScript, HTML, etc…), the ESAPI library will be used to encode the input.
5. The last step is for the XSS Validator to return the resulting string back to the aspect's advice. The advice will then call the proceed() method and pass it the encoded string.

The most difficult aspect of implementing the XSS validator algorithm was to catch all possible exceptions that the ESAPI encoder and validator classes can throw. In the event that an exception occurs, AspectShield handles it gracefully to ensure that no functionality of the web application is broken. The algorithm also supports different types of input such as String and byte arrays in order to support all possible join points identified by the SCA.

5 Validation and Results

This section describes the evaluation of the work. The evaluation will show that AspectShield successfully mitigates both SQL Injection and XSS vulnerabilities without altering source code or breaking application functionality. The evaluation will also show that the libraries and algorithms used to eliminate two of the most important web application vulnerabilities are not only functional but also do not impact application performance by more than an average of 1.99ms per request. Both of the aspects generated by this program will be evaluated in a live environment because they will be built into and executed as part of each of the three case study applications chosen.

5.1 Identifying the Case Study Applications

Since we used AspectJ for AspectShield, our case studies are web applications written in Java. Our other selection criteria for applications included availability of source code, application size of at least 300 classes, support of MySQL or Oracle databases, and developer activity. Websites such as FreshMeat.net, SourceForge.net, and GitHub.com were searched to find suitable candidates.

The first project selected was a popular open source enterprise content management framework called Alfresco. This program has over 140,000 community members, over 2000 enterprise customers, and over 3,000,000 downloads [43]. This application was chosen because as a content management application, it has many points of entry that could potentially be exploited by hackers.

The second project chosen was Apache's OfBiz. This application is one of the Apache Software Foundation's projects and is one of the best open source ERP and E-Commerce implementations. OfBiz was chosen because of its use in E-Commerce, where these types of applications are heavily targeted because they contain personal information such as addresses and credit card information.

The last project chosen was a less well known application called JadaSite, which is another open source E-Commerce framework. This project was chosen because it

makes heavy use of newer web technologies such as AJAX and WYSIWYG user interfaces.

5.2 Methods for Validating AspectShield

We perform three types of validation for AspectShield. First, we evaluate the algorithms and libraries invoked by aspects at each join point. Second, each of the aspects will be evaluated as part of the Alfresco, OfBiz, and JadaSite applications. The first method for evaluation is to determine the functionality of the utility class the aspects call at each join point, the XSS Validator library, and the SQL Injection Validator library. Third, we create unit tests for each of the libraries' methods with JUnit4, and then running a stress test to measure the performance of the libraries.

5.3 SQL Mitigation Algorithm JUnit Results

For the initial set of tests, a list of twenty-one SQL queries was executed fifty times for both the ESAPI MySQL and Oracle encoders. None of the queries exceeded 300 characters. The list contained different types of malicious content that could result in exploits ranging from injection of scripts to bypassing login forms. After the JUnit4 test was executed, the log file that contained the results of each query test was analyzed. The result showed several promising indicators that the desired results were achieved in both successfully mitigating SQL Injection and doing so in less than 5ms on average. The results of the log file provided information such as removal of comments from the query and tautology detection and removal. One such example is on the SQL query "SELECT * FROM Users WHERE ((Username='1' or '1' = '1'))/*') AND (Password=MD5('password')))". This query contains a comment that would bypass the password checking for login validation as well as a tautology, '1' = '1', that would also bypass login checks. This query was successfully mitigated by removing the comment entirely and replacing the tautology with an expression that would evaluate to false.

The second characteristic of SQL Injection Validator execution that was analyzed was the execution time for each of the query validations (Figure 3). Data was collected from the log file, then the maximum, minimum, average, median, and mode were all calculated. The longest execution time was 33ms, and the shortest was 1ms. The average query validation time was only 5.79ms, and the most common time, the mode, was 5ms. The fact that the longest execution period was only 33ms was very encouraging considering that it was much shorter than the average request.

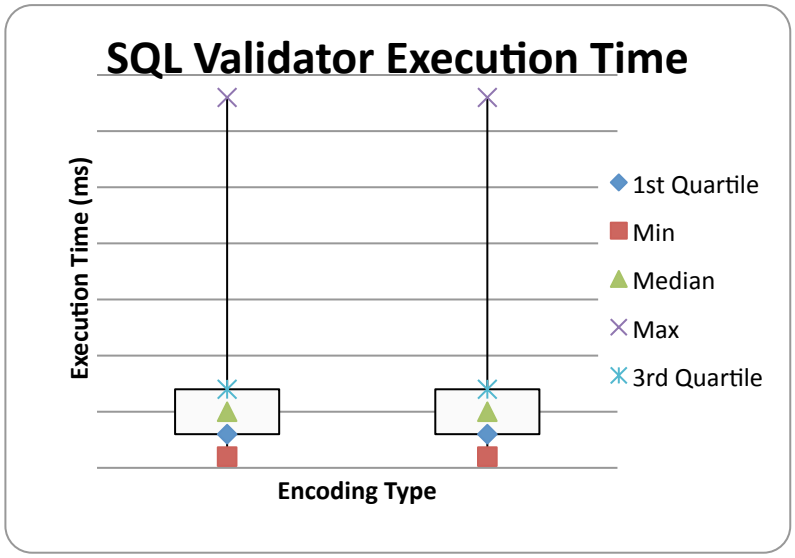

Fig. 3. SQL Mitigation Algorithm execution time

5.4 XSS Mitigation Algorithm JUnit Results

The evaluation results for testing the XSS Validator contained a considerably larger amount of information because of the many different mitigations that are available to the user. However, the 13 different categories can be split into two separate categories that had similar results. The "encoding" category consists of mitigations that use the ESAPI encoder library, which simply encodes the input characters according the context in which they would be used. The second and generally more involved category is "whitelist", where user input is compared to a regular expression string. The results for both can be found in the table below:

Validation Type	*Average (ms)*	*Max (ms)*	*Min (ms)*	*Median (ms)*
HTML Attribute Encode	0.04	13	0	0
Email Whitelist	3.1	197	0	1
Alpha Numeric Whitelist	3.3	199	0	1
URL Encoding	0.09	6	0	0
SSN Whitelist	3.06	199	0	1
Zip Code Whitelist	3.22	196	0	1
Credit Card Validation	4.53	229	0	1
HTML Encoding	0.05	11	0	0
CSS Encoding	0.03	1	0	0
Alpha Whitelist	3.3	209	0	1
Javascript Encoding	0.08	47	0	0
IP Address Whitelist	3.08	198	0	1

As shown above, the items in the "encoding" category had significantly lower average and maximum execution times. The lowest of these was for CSS encoding, which only took a third of a millisecond, and the highest was 47ms for the JavaScript

encoding of a string. The longest execution time belonged to credit card validation, which took almost a quarter of a second at 229ms. The reason that this function takes so long to execute is because the ESAPI validator evaluates the string to see if it matches several different credit card patterns and computes the Luhn checksum, which is used to validate credit card numbers.

5.4 Results of Integration With Case Study Applications

The final part of the evaluation process was to test the aspects with three case study applications running in a live environment. The difficulty of this validation step comes from the fact that the vulnerabilities identified by the SCA are scattered throughout the application and the path to each of these can be difficult to reach since all the projects are enterprise scale applications. Therefore, the chosen approach is to only evaluate the execution of join points that are easily reachable by a typical user of the application. When doing this step, the aspect generation program must either be extracted into a JAR or included in the build path of each of the applications so that the XSS and SQL Injection Validators as well as required libraries such as the JSQL Parser can be referenced.

For the OfBiz project, 22 join points were evaluated and the execution time and result were analyzed. Each join point was able to execute successfully and with similar time to the JUnit4 execution data explained above. Out of the 22 join points evaluated, 16 were possible XSS vulnerabilities with whitelist validation and 6 applied SQL encoding. Since the Alfresco project consists of multiple projects with different contexts, the "repository" project was chosen for execution since it contained a fairly large portion of the vulnerabilities identified. This project contained 11 SQL Injection and 4 XSS vulnerabilities. Both of the aspects executed as expected, except one in the SQL Injection category, where a table creation statement was not supported by the JSQL Parser. However, even though the query was not supported, the program still executed normally, because the SQL Injection Validator returns the original value if query validation fails. The JadaSite project had no issue executing any of the 8 XSS and 14 SQL Injection join points that were tested.

5.5 Evaluation – OWASP Webgoat Project

To provide an extra layer of validation using an application with which most professionals in the web application security field are familiar, AspectShield was applied to the OWASP WebGoat version 5.2 project. WebGoat is a deliberately insecure Java web application that is designed to teach web application security. It contains a number of purposefully implemented vulnerabilities, including several SQL Injection and XSS vulnerabilities.

Fortify SCA was used to locate these vulnerabilities. Using the Aspect Generator, aspects were created to implement mitigations at runtime by intercepting potentially malicious user input. The application was compiled and deployed with the generated aspects, and then each of the XSS and SQL Injection modules were tested.

The two modules tested manually were Injection flaws and XSS. For SQL Injection, several types of queries such as insert, update, and delete were executed as well as inputs with multiples queries and tautologies. All of the queries attempted to execute would normally exploit the application but were successfully mitigated with AspectShield. For XSS vulnerabilities, the fixes specified to the XSS aspect during the generation of the aspects were to do Javascript encoding on various inputs. The aspect successfully mitigated reflected, stored, and DOM based XSS attacks that would normally succeed in the modules associated with that type of vulnerability.

5.6 Evaluation Conclusion

Evaluation showed that AspectShield successfully prevented the exploitation of XSS and SQL INJECTION vulnerabilities in three case study Java web applications, as well as in the OWASP WebGoat application. JUnit testing of both the SQL INJECTION and XSS mitigation algorithms proved that the implementation would not significantly affect application performance or interrupt execution flow. The implementation of the vulnerability mitigation aspects across three enterprise level web applications showed that the implementation can easily be applied to existing code and successfully mitigate attacks.

6 Related Works

6.1 AOP and Security

Security with AOP has been the subject of study in several different publications [6], [7], [8]. Some of the papers that influenced this work include the work done by Robin C. Laney and Janet van der Linden [7], where the authors were able to leverage the power of AOP in order to make significant changes to legacy applications. This was particularly interesting, because often programmers are assigned the task to implement some improvement to a piece of software that has not been modified for several years and has little documentation. The authors showed that programmers can use AOP to evolve legacy code and leave behind digital signatures that reduce the likelihood of breaking existing functionality while enhancing the application overall. In the work done by Minhuan Huang, Lufeng Zhang, and Chunlei Wang [8], they created a fully functional library that implements security features across an application using AOP. Although their library mostly focused on encryption and decryption, it showed how security could be implemented using AOP.

Seinturier and Hermosillo wrote a paper which relates closely to this work [9]. Their tool, AProSec, detects inputs to a web application using aspects to intercept potential XSS and SQL Injection attacks. Their aspects then either warn the user or reject the potentially harmful data input. Their approach was unique in that they wrote aspects that implement security detection functionality in a web application server's native libraries without having to modify the web application server code or write their own. Some of their concepts such as intercepting request and response parameters were

leveraged in the creation of aspects in this research. Another framework created by Zhi Jian Zhu and Mohammad Zulkernine uses AOP for intrusion detection for some of the most common attacks in the Web Application Security Consortium [23]. The AspectShield tool takes this approach in a different direction by intercepting vulnerable code detected by the SCA and than fixing harmful inputs during run time.

While we were making final revisions of this paper for publication, we discovered a paper describing the implementation of a system similar to AspectShield that had been published after we completed the version of this paper that we submitted for publication. This paper describes the creation of an Eclipse IDE plug-in that does automatic discovery of weaknesses in the application code and with the assistance of the developer, remediates them with AOP [24]. The plug in also makes use of the ESAPI libraries created by OWASP and generates aspects based off of user selected validation and encoding techniques.

6.2 Security with AOP for SQL Injection

When looking at the most prevalent web application security vulnerabilities, injection is typically at the top of the list at every reliable source. SQL Injection tends to be the most harmful of these and it is ranked as the second most common form of attack on web applications [10]. One of the most extensive works of research done by V. Shanmughaneethi, Yagna Pravin, and Emilin Shyni uses aspects to analyze a SQL query for potentially malicious content [10]. This tool uses aspects which call web services to analyze queries and create errors in order to prevent malicious SQL from being executed. This is a good approach in theory, but the authors do not specifically discuss the implications of making web service calls with respect to performance and reliability of these web services.

In his book [11], Justin Clarke briefly discusses how AOP can be leveraged to hot-patch applications that are vulnerable to SQL Injection at runtime. He recommends using one of the AOP implementations such as AspectJ and Spring AOP to implement checks for insecure dynamic SQL libraries. Most of the references, such as Clarke's book, offer a few sentences on how the paradigm could be used but do not reference any concrete implementations. Even solutions that do provide concrete implementations, such as the Shanmughaneethi paper, only work as far as identifying vulnerabilities but don't do much to mitigate them.

6.3 Security with AOP for XSS

AOP can be used to mitigate and in some cases eliminate XSS vulnerabilities in web applications. This is especially true when the application in question would require a complete re-write in order to achieve security [23]. According to OWASP, the best two ways to prevent XSS is to escape all untrusted data based on the content of the web page and to do whitelist validation on user inputs [21]. Using AOP, a developer can create aspects to intercept incoming and outgoing data that would be displayed to the user and apply either escaping or whitelisting without modifying the existing source code. Mece and Kodra [24] were able to create a XSS validation aspect that does

whitelisting of user inputs. Their "validator" aspect treated all strings that were not alphanumeric as potentially dangerous and denied them. While this is certainly a very safe approach, applying such restrictions onto an existing application would almost certainly break functionality because many applications require inputs much more complicated than just an alphanumeric string.

There have been multiple studies with the intent to use AOP to eliminate XSS vulnerabilities [28] [29] [30]. However, most of these simply discuss the idea of using aspects in order to achieve security and very few have working implementations. Of the papers with implementations, the implementations are simple ones such as regular expression whitelisting of input that just demonstrate the potential usefulness of the AOP paradigm without offering in-depth solutions.

7 Future Works

There are a number of features and improvements that can be added to AspectShield to make it a more effective security tool and provide a better overall user experience. The first is to extend the program to support mitigation of additional types of vulnerabilities. We intend to examine the possibility of adding additional mitigations for the remaining vulnerabilities of the OWASP Top 10. Along with support for mitigation of a wider range of threats, it would also be helpful to create a graphical user interface for the aspect generator program in order to improve user experience.

A second potential area of improvement is to extend AspectShield to other programming languages. One challenge will be finding a suitable AOP implementation for each additional programming language to be supported. Since the implementation for each language would be different, new template files, function definitions, and libraries for mitigating vulnerabilities would need to be created for each programming language.

A third area for future work would be to extend the program to support multiple static analysis tools. Different automated static analysis tools find different vulnerabilities in source code. Additional plans for future work include an Eclipse plug-in that does the mitigation aspect generation automatically, and a multi language API for creating all parts of the security aspect generation process.

8 Conclusions

Two of the most common vulnerabilities in web applications are SQL Injection and Cross-Site Scripting. Thousands of web applications process personally identifiable information such as SSNs, credit card numbers, and addresses every day, and many of these applications have a significant number of SQL INJECTION and XSS vulnerabilities that can be exploited by a malicious user. The AspectShield tool described in this paper creates aspects that prevent malicious content from being executed or stored in Java web applications using the results of the Fortify Source Code Analyzer and the users' choice of mitigation technique. The most significant

feature of the approach identified is creation of the AspectShield tool, which does mitigation of vulnerabilities without the need to modify potentially fragile source code.

Our approach was to apply the AOP paradigm to execute validator classes at the locations of the vulnerabilities identified by Fortify SCA. This modular approach to implementing security allows the developers to use separation of concerns where they can apply any future security algorithms to a single location. When the user executes AspectShield, they will be given a choice of fixes to implement for each of the vulnerabilities detected by the static analysis tool. The two resulting aspects, one for XSS and the other for SQL Injection, contain pointcuts and advices that will isolate join points throughout the applications' source code and weave in the necessary code that will ensure the mitigation of these vulnerabilities.

In order to evaluate the success of the aspects created, WebGoat and three enterprise level open source Java web applications were chosen as case studies. These applications are from the E-Commerce, content management, ERP, and document management categories. AspectShield generated vulnerability mitigation aspects based on static analysis of these applications. Evaluation consisted of unit tests using the JUnit testing framework, integration of aspects into each project, and testing the mitigation aspects as part of the running applications. The evaluation proved that each of the aspects not only mitigate XSS and SQL INJECTION attacks but also do it very efficiently with most execution times being less than 10 milliseconds. The low execution time of the aspects' at each join point is significant because it is very important that the introduction of the security code did not heavily affect the execution time of the original applications.

In conclusion, this paper describes the implementation of a program that generates XSS and SQL INJECTION mitigation aspects that can be applied to mitigate vulnerabilities in both new and legacy web applications using information from static source code analysis. The main advantage of this approach compared to others evaluated that it does not require any modification to legacy code and provides a centralized location for the application's security logic. The evaluation of generated aspects with three enterprise level projects provides a great level of confidence that the approach is both valid and effective at mitigating some of the most prevalent threats to web application security.

References

1. Webroot. State of Internet Security – Protecting Enterprise Systems [Whitepaper] USA: Webroot Software Inc., 2007.
2. Electronista. "LulzSec hacks Sony Pictures, reveals 1m passwords unguarded." Electronista Media Inc., 2 Jun. 2011.
3. "Measuring Website Security: Windows of Exposure." WhiteHat Website Security Statistic Report.,14 Mar. 2011.
4. V. Shanmughaneethi, Ra. Yagna Pravin, C. Emilin Shyni, S. Swamynathan: SQLIVD - AOP: Preventing SQL Injection.
5. OWASP (Open Source Web Application Security Project) . OWASP Top 10 – 2010 Edition. OWASP Foundation, 2010.
6. "Fortify Source Code Analyzer – Capabilities." HP Fortify. Web. 2011.
7. Laddad, Ramnivas. "AOP @ Work: AOP Myths & Realities." IBM Developer Works, 14 Feb. 2006.

8. ESAPI Interface Encoder. The Open Web Application Security Project. Web. 2011.
9. ESAPI Validator Library. The Open Web Application Security Project. Web. 2011.
10. Li, Sing. AOP: Patching in the 21st Century. Developer Fusion. Web. 23 Jul. 2010.
11. Bostrom, G. Database Encryption as an Aspect. Proceedings of AOSD 2004 Workshop on AOSD Technology for Application level Security., Mar. 2004.
12. Laney, R., van der Linden, J., Thomas, P. Evolution of Aspects for Legacy System Security Concerns. Proceedings of AOSD 2004 Workshop on AOSD Technology for Application level Security., Mar. 2004
13. Huang, M., Wang, C., Zhang, L. Toward a Reusable and Generic Security Aspect Library. Proceedings of AOSD 2004 Workshop on AOSD Technology for Application level Security., Mar. 2004.
14. Hermosillo, G., Gomez, R., Seinturier, L., Duchien, L. Using Aspect Programming to Secure Web Applications. Journal of Software, Vol. 2, No. 6., Dec 2007.
15. Clarke, Justin. SQL Injection Attacks and Defense. 1st ed. Syngress, 13 May 2009. 1 Mar. 2011.
16. Mece, Elinda. Kodra, Lorena. Towards full protection of Web Applications based on Aspect Oriented Programming, pp. 33-37, GJCST 2012.
17. Arthur, Charles. "Twitter users including Sarah Brown hit by malicious hacker attack." Guardian News. 21 Sep. 2010.
18. Win, Bart ., Viren Shah, Wouter Joosen, and Ron Bodkin, editors. *AOSDSEC: AOSD Technology for Application-Level Security*, March 2004.
19. Bodkin, Ron. "Enterprise Security Aspects." *AOSDSEC: AOSD Technology for Application-Level Security*. Ed. Bart . Win, Viren Shah, Wouter Joosen, and Ron Bodkin, March 2004.
20. Fortify. Leading Bank Turns Security into a Differentiator with Fortify SCA. Fortify Software Inc. 2008.
21. Feathers, Michael, Working Effectively with Legacy Code, Prentice Hall, 2004.
22. Higgins, Kelly J. "The Cost of Fixing an Application Vulnerability". Security Dark Reading. 11 May 2009. < http://www.darkreading.com/security/news/ >
23. Zhi Jian Zhu , Mohammad Zulkernine, A model-based aspect-oriented framework for building intrusion-aware software systems, Information and Software Technology, v.51 n.5, p.865-875, May, 2009
24. Gabriel Serme, Anderson Santana De Oliveira, Marco Guarnieri, Paul El Khoury. Towards Assisted Remediation of Security Vulnerabilities. 6th International Conference on Emerging Security Information, Systems and Technologies, August 2012.