ABSTRACT OF THESIS

Eliminating SQL Injection and Cross-Site Scripting With Aspect Oriented Programming

Cross Site scripting (XSS) and SQL injection are two of the most common vulnerabilities found in applications. According to a study done by the Web Application Security Consortium (WASC) on 12,186 web applications, the percentage of sites with these vulnerabilities is 38 and 13 percent, respectively [3]. The fundamental reason these vulnerabilities exist in web applications are critical design flaws which lead to security issues across entire projects. It is typical in the case of web applications that developers continue to write insecure code and only fix these issues when they are noticed or become a problem [4]. Using Aspect Oriented Programming (AOP), modules can be created to address these security vulnerabilities across an entire application without modifying existing source code. This paper will explain in detail how the use of AOP and AspectJ in particular can be leveraged to create a tool for eliminating these two major security vulnerabilities in open source Java web applications. The implications of a tool for successfully eliminating these vulnerabilities would lead to a significant improvement in web application security by uncovering fundamental design flaws, providing sanity checks for programmers and architects, and make applications compliant with PCI DSS Standard with respect to XSS and SQL Injection.

Eliminating SQL Injection and Cross-Site Scripting With Aspect Oriented Programming

by

Bojan Simic

Northern Kentucky University

May 2012

Eliminating SQL Injection and Cross-Site Scripting With Aspect Oriented Programming

THESIS

A thesis submitted in partial fulfillment of the
requirements for the degree of Master of Science in Computer Science
at Northern Kentucky University

By

Bojan Simic
Highland Heights, Kentucky

Director: Dr. Maureen Doyle
Associate Professor of Computer Science

Highland Heights, Kentucky

2012

ACKNOWLEDGEMENTS

Fortify Source Code Analyzer is a product owned and licensed by HP.

Java is a registered trademark licensed by Oracle.

AspectJ is a registered trademark of the Eclipse Foundation.

The OWASP Enterprise Security API (ESAPI) is licensed by OWASP.

TABLE OF CONTENTS

TABLE OF FIGURES

Chapter 1 Introduction

1.1 Introduction

Insecurity is a fact in today's Internet. The most private information such as credit card numbers, social security numbers, and personally identifiable information (PII) is transferred across the Internet on a regular basis. The statistics on Internet security tell us that one in three computers fall victim to viruses [1]; 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 [2]. As the Internet evolves, the variety of attacks evolves with it, and Internet security becomes a moving target that is becoming more difficult to hit.

Lack of Internet security cannot be attributed to any one particular factor. Instead, it is a very complicated and expensive task that often requires considerable effort and knowledge. In order to have a secure web application, there needs to be a rigid security-oriented architectural design, routine maintenance, code reviews, and extensive penetration testing. Unfortunately, software security is often seen as an afterthought in the software architecture design and this often leads to the majority of vulnerabilities. Many businesses are simply not willing to pay the overhead costs associated with operating a secure application. This often leads to putting an essentially ineffective patch on an architecturally insecure product in order to have some form of "security". These are the kind of solutions that result in the unsatisfactory statistics above.

Implementing software security is a very important but complicated task for any development team. When architecting a piece of software it is imperative that the method chosen for implementing security is simple, effective, robust, modular, and scalable in order for the solution to be maintainable. This paper will discuss a programming paradigm with all these values and how it can be implemented in new and existing web applications to mitigate some of the most common vulnerabilities in Java web applications.

In the remainder of this chapter, we will discuss the motivation for the writing of this thesis, the background research that was done in preparation, and how the remainder of the paper is laid out.

1.2 Motivation

According to White Hat Security, “The average website has 230 vulnerabilities that are in the HIGH, CRITICAL, or URGENT state according to the Payment Card Industry Data Security Standard (PCI-DSS) for software security”. [2] This statement serves as the one of the primary motivations for this thesis. When looking at the state of Internet security, it is important to remember that not all vulnerabilities are created equal. For example, Cross-site Scripting (XSS) and SQL Injections (SQLI) are often the vulnerabilities with higher priority because of the potential repercussions that could arise if they are exploited. When looking at the statistics, 64% of websites have XSS vulnerabilities compared to 14% that have SQL Injection bugs [2]. Although the percentage of applications with SQL Injection issues is lower, a vulnerability of this type can potentially lead to an entire database worth of stolen or lost information [4]. The prioritization of vulnerabilities across multiple industries such as financial, retail, and telecommunications is what motivated the writing of this paper to focus on SQL Injection and Cross-site Scripting in particular.

Another major motivating factor is the current state of software security programming practices. When a software developer researches software security best practices, they are often given instruction to do things such as input validation, sanitizing data, avoid using dynamic SQL, etc… While all of this information is relevant and helpful, often times there are no ways to enforce these best practices in a real world software development environment where the programmer has pressures from strict deadlines. The motivating question here is “what is a time and cost effective way for a development team to create a secure product?” The key to this is simplicity because the simplest, most unobtrusive approach must be taken in order to realistically create a secure product. This is the motivation for using the Aspect Oriented Programming

(AOP). This paradigm allows the developer to use a modular approach that is able to cut across multiple concerns in order to implement security. This avoids the often scattered and tangled security code to be in a single module that is applied to any part of the software where needed.

The third major motivating factor for this thesis is the need to reduce time and cost to implement security into legacy web applications. Although it is always preferred to implement security at every stage of the development process, when an application is already deployed or security was not an absolute necessity in the first place, there needs to be a time and cost effective way of implementing it. We typically try to avoid the “penetrate and patch” approach to security but the simple fact of the matter is that flaws are usually discovered in the production environment. When dealing with this scenario, AOP is an obvious choice to implement security because the developers do not have to make changes to existing source code. Instead, they can use AOP’s joint-point model to secure existing source code and even compiled third party libraries that may need security patches.

1.3 AOP and the AspectJ Implementation

AOP is the programming paradigm used to implement separation of concerns in software. It uses the concepts of both procedural and object-oriented programming (OOP) used for decades to increase modularity by allowing all concerns that affect multiple parts of the application to be centralized. AOP provides the structure for applying cross-cutting concerns across complex software projects in the way that OOP provides features such as inheritance. This concept provides easier maintenance, readability, and reusability of code because it is in a centralized place. The individual, modularized portion of code that does cross-cutting is called an aspect.

AspectJ is the extension of the Java programming language that implements aspect-oriented programming. This extension allows for definition of code to execute at pre-defined points at program execution time. These join points are the defined points of the program’s execution that are grouped as pointcuts that also refer to values at individual join points. The concept of an advice allows for definition of additional code to be executed before, after, or around the join point to add to its behavior. The totality of the aspect consists of one or more pointcuts, an optional constructor, member declarations, and an advice for each pointcut.

The Joint Point Model is the critical piece of the AOP mechanism. Each join point acts as a reference to a part of a complex program that can be acted on. Each pointcut that evaluates to a potential collection of joint points is defined by several methods:

- Method or constructor call – can intercept the method or constructor in any object defined.
- Method or constructor execution – when the method or constructor in an object is invoked.
- Accessors or mutators – can intercept the setting or getting of a field in a class, object, or interface.
- Execution of exception handlers.
- Class or object initialization.

The pointcuts can be defined using pointcut designators that reference parts of code such as class names, method names, method signatures, method return types as well as wild card characters to get the desired resulting join points.

The Advice portion of the aspect is a piece of code similar to a method that will be executed at each of the join points identified by the corresponding pointcut. AspectJ supports three types of advice: before, after, and around. If the advice is of type before or after, the code declared will be executed either before the join point or after it is finished executing. This is typically used to log certain events or to do any necessary pre or post processing of data for a particular module. The around advice is used to intercept the normal execution of the join point. The advice allows for actions such as validation of input or output of the join point as well as catching of any exceptions that may occur.

The AspectJ model of behavior defines the hierarchy in the aspect for each advice. The around advice is executed first in order of most-specific first, then the before advice, and finally the after advice (least specific first). This hierarchy allows for error free execution of cross-cutting concerns even in the event that multiple aspects advice upon the same join point. The aspect definition is similar to that of a regular Java class and can have its own constructor (with no parameters) and attributes. The AspectJ Compiler (AJC) is used to do aspect weaving that ensures the execution of every advice at individual join points. Each piece of the program source code is compiled as normal when there is no join point identified and static points that represent the advice logic are inserted wherever join points are defined.

The AspectJ implementation of AOP is the top rated extension for the Java programming language. The extension provides the benefit of more modular and concise code; a structured framework for implementing cross-cutting concerns; programming tools and extensions for programming environments; and code readability because logic that is typically boilerplate code can be extracted into individual modules. Therefore the goal of this thesis is to leverage this extension to apply it to eliminating SQL Injection and XSS vulnerabilities in Java web applications.

1.4 Background Research and Related Work

1.4.1 Security and the Aspect-Oriented Programming Paradigm

There are many ways that people are approaching web application security but the only thing for certain is that there is no silver bullet solution to the problem. However, since the best solution is usually the simplest one, the AOP paradigm tends to be a more suitable candidate to at least mitigate the security problem in web applications. By applying security with AOP and the joint-point model, the development team has the option to modularize their security code and have it apply across several concerns such as the web application server and the database layer of the application. This is the main reason to use AOP for security since there is usually very little reason to include security based logic into the core business logic of the application. The development lead can also apply various design patterns such as factory in order to enforce certain coding restraints for security.

Security with AOP has been the subject of study in several different publications [6], [7], [8]. Some of the ones that influenced this paper 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 using AOP programmers can 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 were able to create a fully functional library that implements security features across an application using AOP. Although their library mostly focused on the encryption and decryption aspect of software security, it showed how similar works could be implemented for other security issues such as the OWASP top 10.

Seinturier and Hermosillo have done one of the most interesting works in the area of AOP and Security and it touches very closely to the topic of this research [9]. Their tool, AProSec detects inputs to the web application by 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 the fact that they wrote aspects that implement security functionality to the web application servers' native libraries without having to modify the web application server code or write their own. There is no doubt that theirs is a very good tool to have and some of their concepts such as intercepting request and response parameters were leveraged in the creation of aspects in this research.

What we've learned on the topic of using AOP for security is that it can be a powerful tool to leverage in the hope to achieving a secure web application. The AOP paradigm can be used to implement security in legacy code, create a modularized approach to web application security, and allow developers to use a common approach to implementing security ranging from broken authentication/session management to SQL Injection and XSS.

1.4.2 Aspect-Oriented Programming and 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]. The idea of SQL Injection is basically the input of malicious code that results in the execution of a query that could either steal, or delete important information. There have been many works on preventing SQL Injection but very few that use the AOP paradigm.

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 the book by Justin Clarke [11], he 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 (which is used for this paper) and Spring AOP to implement checks for insecure dynamic SQL libraries. By searching different libraries and online journals, it quickly became apparent that there is not much research done in preventing SQL Injection with AOP. Most of the resources such as the Clarke's book only offer a few sentences on how the paradigm could be used but do not reference any concrete implementations. Even those solutions that do provide concrete implementations such as the Shanmughaneethi paper, only work as far as identifying vulnerabilities but don't do much in the way of fixing them.

Though there is a lack of sources for implementing security with AOP and SQL Injection, there were several resources that could be referenced in the SQL Injection prevention effort. The creation of the Intrusion Detection and Prevention System (IDPS) by Varian Luong [12] provided great insight into the different types of SQL Injection attacks that a programmer has to keep in mind when creating a secure application. The tool did detection of the threats and emailed the system administrator as it deterred attackers. Although this approach is preferred in many scenarios, the IDPS has to be deployed and coded as part of the overall application and does not seem to scale incredibly well because it would require integration into every new module as it is developed.

There are also other techniques that are often used to implement security against SQL Injection attacks. These include Safe Query Objects [13] that encapsulate all database queries in objects and provide a secure and reliable way of executing database queries. Another similar approach, SQLDOM [14] uses a strong link to the specific database schema in order to enforce security by only creating queries from strongly connected classes. In the work done by V.B. Livshits, the author uses static analysis to detect SQL Injection vulnerabilities in java applications [15]. Livshits' research influenced the need for a powerful static analysis tool in order to detect vulnerabilities and highlighted some of the attributes (i.e. detecting false positives) such a tool would be required to have. In G. Wassermans' research [16], the authors developed a framework for detecting tautologies in a database query. This approach is very helpful but does not prevent other types of attacks and is usually an incomplete solution in the scenario where the tautology is complicated or uses a database specific function. For example, "RAND() > .01" is not technically a tautology, but it most likely will evaluate to true anyway when the query is executed on the database because the random number generated will most likely be greater than .01.

The use of static analysis combined with runtime validation is the latest set of efforts to detect and prevent SQL Injection attacks. One tool, WebSSARI [17], uses logic to determine whether or not it should do checks for invalid input either at compile or run time. It then analyzes the users' input and throws an error if an attack is detected. While WebSSARI is only compatible with the PHP language, it certainly shows promise because it attempts to prevent attacks at different stages of the applications execution. Another modern approach was to create a parse tree of the initial database query and combine it with the parse tree of the resulting query and then compare the two for discrepancies. This approach seems to be very effective but requires the developer to know the exact state of the query before and after any user input which can be complicated when there is complex logic surrounding the building of the query [18]. Another drawback to this approach is that it cannot possibly be used on any legacy applications where the code uses dynamically structured queries because it would not be possible to know the before and after states of the queries. There have been several other studies that use a Non-Deterministic Finite Automaton to create models for creating possible versions of a static query. This approach while very effective, it was only proven on very small projects and had no guarantee of the percentage of false positives [19] [20].

All of the above efforts to eliminate or mitigate SQL Injection attacks work in some capacity. The one area where they all fall short of expectations is in the fact that they all require fairly extensive modification to the web applications source code. The ones that require minimum changes to the source code often require version specific patches to the application server or the database definitions. The study of these related works in preventing SQL Injection have solidified the theory that AOP is the ideal paradigm to use for creating a truly secure web application.

1.4.3 Aspect-Oriented Programming and XSS

Cross-site scripting (XSS) is the most prevalent web application security issue [21]. The three known types of XSS attacks are stored, reflected, and DOM based XSS and their main purpose is to steal the users' credentials via cookie hijacking. The stored XSS attack is typically injected and then kept inside the database of the vulnerable application. The injected code is then displayed to various users on a site with forum, comment, message, etc functionality. When the victim application displays the malicious code, a script is typically executed to steal the users' browser data which often includes email addresses, session variables, and even security keys. Reflected XSS attacks are often executed by the user being tricked into clicking a malicious link on a website or in an email. The code that is contained in the link or form is then sent to the vulnerable web server and the user ends up being a victim because the user thinks the code originated from a trusted source. DOM Based XSS is different from the first two because it does not try to change the HTTP response of the application but instead it tries to modify the DOM. The malicious code is passed in as a URL parameter and then used to modify the DOM of the client [22].

There are numerous reasons for considering the elimination or mitigation of XSS in this paper. One prime example is the September 2010 attack on Twitter where the attacker (a 17 year old) was able to exploit the “onMouseOver” event to steal information of many users [25]. In October 2010, the American Express website was under attack and the resulting vulnerability was XSS; the company’s 6th in recent history [26]. While there are thousands of XSS attacks every day, it is important to remember some statistics on attacks that affect applications that transfer some of our personal data daily. According to the Web Hacking Incidents Data Report of 2010, 26% of attacks on the attacks on government succeeded in defacing that organization, 64% of attacks on the finance industry resulted in monetary loss, and 27% of attacks on the retail industry resulted in credit card leakage [27]. This is particularly disturbing to most people who might do online banking and purchasing considering that over a quarter of all attempts on seizing their most valuable data were successful.

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 most 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. The ones that do have implementations, they are often very simple that only show the potential usefulness of the AOP paradigm and do not offer any in-depth solutions. This was a motivating factor in the writing of this paper because the intended outcome is to create a robust program that applies security to any existing Java web application.

1.4.4 Static analysis tools and the Fortify SCA.

In the effort to eliminate or mitigate two of most dangerous threats to web applications, it is a necessity to be able to detect where the vulnerabilities are located in the source code. One of the best tools for source code analysis is Fortify, the winner of the 2011 CODiE awards for “Best Security Solution” [32]. This tool is the vendor of choice for a large number of Fortune 500 companies in the energy, financial, healthcare, e-commerce, media, and several other major industries. Fortify has dozens of case studies on their website that highlight some of their success in the software security industry. One such case study, at a large international bank, showed how Fortify can be successful on large scale projects that exceed 50 million lines of code [31]. The reason for choosing Fortify for this thesis was its success in real world applications, its recommendations by OWASP, and it was a commercial tool available to the research students and faculty at NKU. Fortify was used to analyze three different open source Java web applications for vulnerabilities that would then be mitigated using the AOP approach.

1.5 Thesis Statement

The question that we want to answer in this thesis is the following:

> **Can a tool that mitigates SQL Injection and Cross Site Scripting vulnerabilities in Java web applications be created while maintaining performance and without modifying source code?**

While working to answer this question, several others must be asked and addressed accordingly:

- What programming techniques must be used to maintain the same source code? The Aspect Oriented Programming paradigm must be fully leveraged in order to not modify existing source code, have all required code in a centralized place, and to decrease the possibility of introducing bugs [5].
- How can we minimize the number of false positives? There are instances where a valid input could be seen as a threat. These scenarios must be handled with care so that existing functionality does not break.
- How will the tool prioritize detected vulnerabilities? The tool must be able to determine whether a vulnerability will be addressed based on its potential performance impact and threat level.
- What are the implications of using code weaving to mitigate vulnerabilities? Without visible changes to the original source code, could it become increasingly difficult to debug or could it also help decouple security oriented code out of the core business logic? A goal will be to weigh the benefits of this technique based on code maintainability and readability.
- How will contextual output be encoded and escaped? User input can vary in context and a goal will be to identify the context and use the correct technique to escape or encode the values.
- What methods must be used or created to eliminate SQL queries that are not secure? An important piece will be to create algorithms for rewriting insecure dynamic queries. Will these algorithms need to be created from scratch or will they leverage existing query parsing tools? And if so, which ones offer the best performance?

1.6 Overview

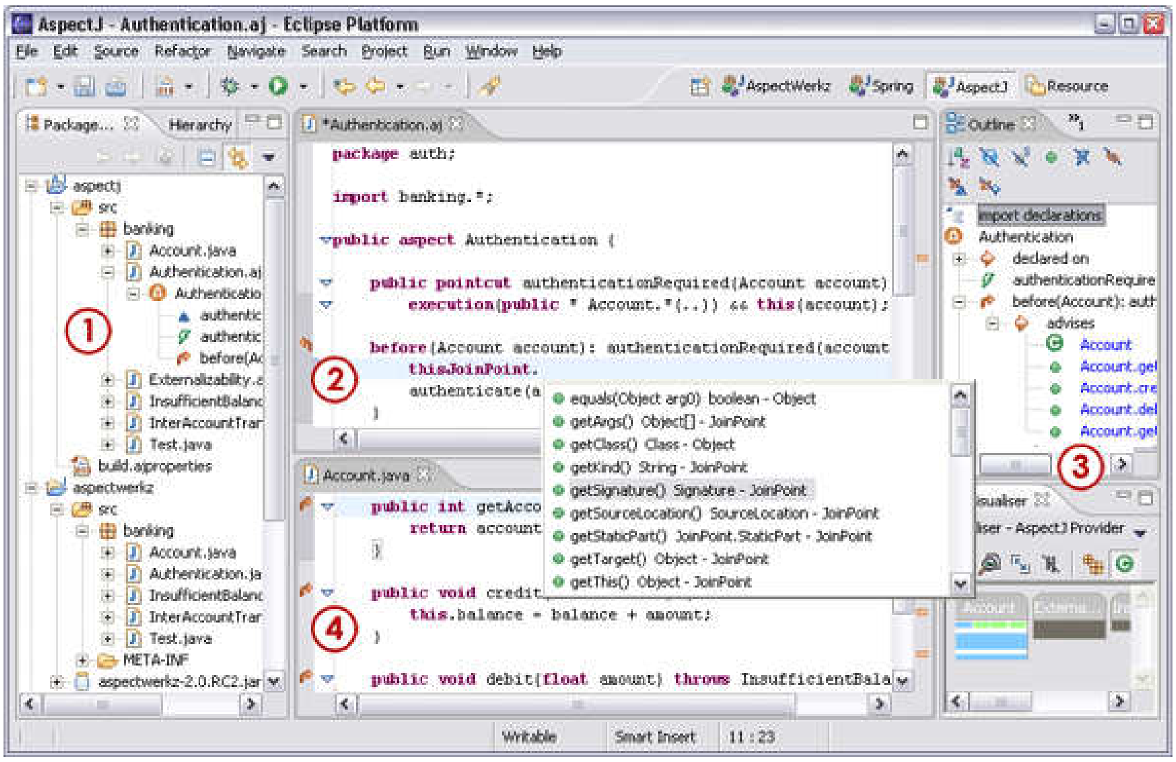
The remainder of this thesis is structured as follows. Chapter 2 will go into detail on the approach for mitigating SQL Injection and XSS in Java Web Applications. It will discuss the strength of the approach as well as potential weaknesses. Chapter 3 will go into the implementation of the program written to achieve the task and any specific algorithms necessary. Chapter 4 will analyze the results of the efforts and cover evaluation of the resulting program. Chapter 5 will summarize and conclude the text as well as cover the intent and planning for any future works.

Chapter 2 Approach

2.1 Choosing the AOP Library

There are several different open-source AOP libraries available for the Java programming language. Some of these include AspectJ, AspectWerkz, JBoss AOP, and Spring AOP [33]. The three main categories that these libraries fit into are XML based, annotation based, and language extension based. One of these, AspectJ is both annotation based and has a language extension capability. The approach to determining which one to use was influenced by the amount of documentation available, real-world implementation, and learning curve associated with using that particular library with an existing project.

AspectJ was the first tool evaluated for the approach and ultimately the AOP implementation used in this paper. It was chosen because of its Java language extension capability that results in a very intuitive way of writing aspect code. The tool also has a very robust Eclipse IDE plug-in (AJDT) since it is a child project under the Eclipse open source umbrella. The plug-in has several extremely helpful features that make implementing AOP and cross cutting concerns much easier due to a very advanced user interface. With this interface (as described in figure 1 [34]), the user is able to view exactly where their aspect is and what parts of the code it’s affecting. In (1) the defined aspects are marked by specific color and text which can be expanded to display the names of the different pointcuts and advices’ that are defined in each particular aspect. In (2) and (3) there is a red arrow that displays to the programmer the different areas of the source code that a particular advice is going to act upon. The user can drill down and inspect each piece of source code and then modify the pointcut if they want to add or remove the code the advice will act upon. If the user is browsing through source code, the AJDT plug-in will give them an orange arrow to signal that an aspect is affecting that particular function call as shown in (4). The AspectJ project has a very rich history from the time when it was first created at the Xerox Palo Alto Research Center (PARC) to the integration into the eclipse project in 2002. The project continues to be developed regularly and interest and implementation has been increasing exponentially [35].



```
package auth;

import banking.*;

public aspect Authentication {

    public pointcut authenticationRequired(Account account)
        execution(public * Account.*(..)) && this(account);

    before(Account account): authenticationRequired(account
        thisJoinPoint.
        authenticate(a
    }
```

```
equals(Object arg0) boolean - Object
getArgs() Object[] - JoinPoint
getClass() Class - Object
getKind() String - JoinPoint
getSignature() Signature - JoinPoint
getSourceLocation() SourceLocation - JoinPoint
getStaticPart() JoinPoint.StaticPart - JoinPoint
getTarget() Object - JoinPoint
getThis() Object - JoinPoint
```

```
    public int getAcco
        return account
    }

    public void credit
        this.balance = balance + amount;
    }

    public void debit(float amount) throws InsufficientBala
```

Figure 1 - AJDT Plugin Interface Example

AspectWerkz is another lightweight, open-source AOP implementation for Java. This library was soon removed from consideration since it merged with AspectJ mentioned above in 2005 [36]. The resulting product was AspectJ 5+ which contained support for annotations and other features from AspectWerkz such as faster load time weaving and Java 5 support. There was strong consideration given to the JBoss AOP project because it had IDE support and an extensive library for implementing aspects. However, the project does not seem as mature as the AspectJ and AspectWerkz combined implementation. The Spring AOP implementation is also robust and effective but it was found that there is significant effort required to implement it outside of the Spring framework. After evaluating multiple implementations of AOP for Java, AspectJ proved to be the logical choice based on all the features and simplicity of implementation.

2.2 Evaluation and Selection of Projects for Case Studies

In order to create a program that mitigates SQL Injection and XSS vulnerabilities, it is necessary to prove the success of the implementation on real world, enterprise level Java web applications. It was necessary to use open source web applications because they offer fully available source code that could be analyzed. The three applications chosen to act as case studies for this project were:

1. Alfresco – an enterprise content management system with over 670 weekly downloads from SourceForge.net [37].
2. Apache OFBiz – an open source ERP, CRM, E-Commerce, SCM, MRP, POS, etc… framework hosted on FreshMeat.net and Github.com
3. JadaSite – an open source E-Commerce implementation in Java with over 140 weekly downloads from SourceForge.net [38].

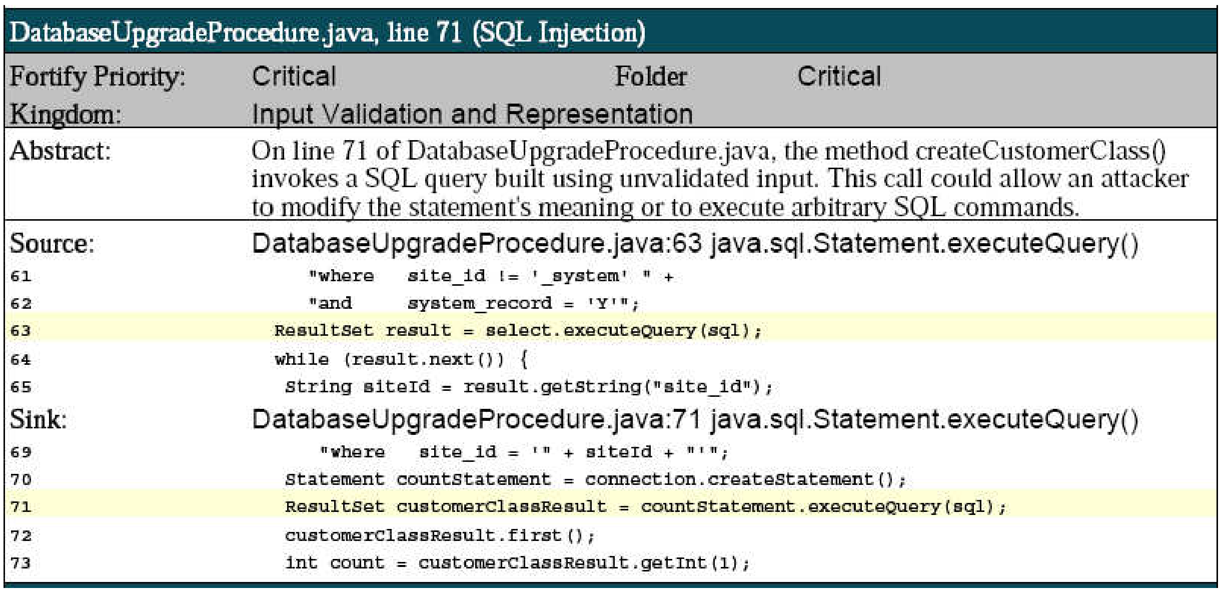
Alfresco was chosen because it is an enterprise level application with many modes of user input, which results in many potential SQL Injection and XSS vulnerabilities. The application is built using Java JDK 5, uses a MySQL Database, and is typically deployed on a Tomcat 5.0+ web application server. The project also has a very active community that makes updates and enhancements on a daily basis as well as a commercial version of the product with support from the Alfresco team of developers. The application has thousands of classes and makes use of many Web 2.0 technologies such as CSS3 and AJAX.

The Apache OFBiz project is one of the largest and popular open source applications available. The application is considered one of the top E-Commerce frameworks used by many large corporations and is considered one of the top 10 ERP implementations available [39]. This Java application can be deployed on multiple application servers and is database agnostic meaning that it can use any database that has a JDBC driver associated with it. The E-Commerce industry is often victim to many attacks because it deals with PII such as physical addresses, email addresses, and credit card numbers of millions of people daily. OFBiz was considered for this thesis because it’s important to prove that vulnerabilities can be mitigated in a large scale ERP and E-Commerce platform.

The JadaSite application is more of a pure E-Commerce platform built on Java that offers multi store support with capabilities to support multiple languages and datasets. This project was chosen in order to ensure support for mitigating vulnerabilities where user inputs can contain different character sets.

2.3 Approaches for Static Code Analysis

The Static Code Analysis (SCA) tool used for this project was the Fortify SCA. One of the reasons this tool was selected is because of its extensive use in the work done at NKU on analyzing vulnerabilities in 14 different PHP web applications [40]. Another major reason is that Fortify is one of most robust SCA tools available. The tool also has an Audit Workbench program that allows the user to use the result of the SCA to audit the vulnerabilities detected. The workbench also has a very useful export feature that allows the user to get an XML or PDF version of the report. The report contains some very useful information such as the location of the vulnerabilities (line number and file), the priority, abstract definition, and category (see figure 2 below). The report also contains the source and sink locations for each of the vulnerabilities detected. The vulnerability source section refers to the location of the code where the vulnerability occurs and the sink refers to the root cause of the vulnerability.



DatabaseUpgradeProcedure.java, line 71 (SQL Injection)

Fortify Priority:	Critical	Folder	Critical
Kingdom:	Input Validation and Representation		

Abstract: On line 71 of DatabaseUpgradeProcedure.java, the method createCustomerClass() invokes a SQL query built using unvalidated input. This call could allow an attacker to modify the statement's meaning or to execute arbitrary SQL commands.

Source: DatabaseUpgradeProcedure.java:63 java.sql.Statement.executeQuery()

```
61          "where   site_id != '_system' " +
62          "and     system_record = 'Y'";
63      ResultSet result = select.executeQuery(sql);
64      while (result.next()) {
65       String siteId = result.getString("site_id");
```

Sink: DatabaseUpgradeProcedure.java:71 java.sql.Statement.executeQuery()

```
69           "where   site_id = '" + siteId + "'";
70       Statement countStatement = connection.createStatement();
71       ResultSet customerClassResult = countStatement.executeQuery(sql);
72       customerClassResult.first();
73       int count = customerClassResult.getInt(1);
```

Figure 2 - Vulnerability Description from Fortify Report

The Audit Workbench program's ability to export the report in multiple formats such as XML made it more convenient to analyze the data with an XML parser available to many programming languages such as Java. The Fortify SCA is also used and recommended by Brian Chess and Jacob West, the authors of "Secure Programming with Static Analysis" [41]. Using this tool was the logical choice since it was proven to be effective in detecting major vulnerabilities such as SQL Injection and XSS on many projects in several programming languages.

2.4 Mitigation Technique Selection

Before deciding on which approach to use for SQLI and XSS vulnerability mitigation it is important to analyze several different techniques and determine which are best suited for an effective and efficient AOP security implementation. The various techniques analyzed come from OWASP because it is an authority on web application security and a comprehensive source for eliminating or mitigating the top ten threats.

The three primary defenses against SQL Injection are the use of parameterized queries, stored procedures, and escaping user input. An additional defense is to do white list input validation in order to prevent input of incorrect format to be added to the query. For the AOP implementation, the options chosen for mitigating SQL Injection are to do escaping of all user supplied input and perform white list validation. The choice to escape user input comes from the fact that it is the most straightforward approach that does not require the modification of legacy code. Aspects can be written to intercept and analyze the query and then escape all expressions before it is executed. The approach to use prepared statements or stored procedures aren't feasible because they would require extensive rewriting of dynamic query code and therefore modifying existing source code and possibly introducing bugs or unwanted behavior. The escaping method is also preferred because OWASP provides a security encoding library for the Java programming language and support for multiple database types such as MySQL and Oracle.

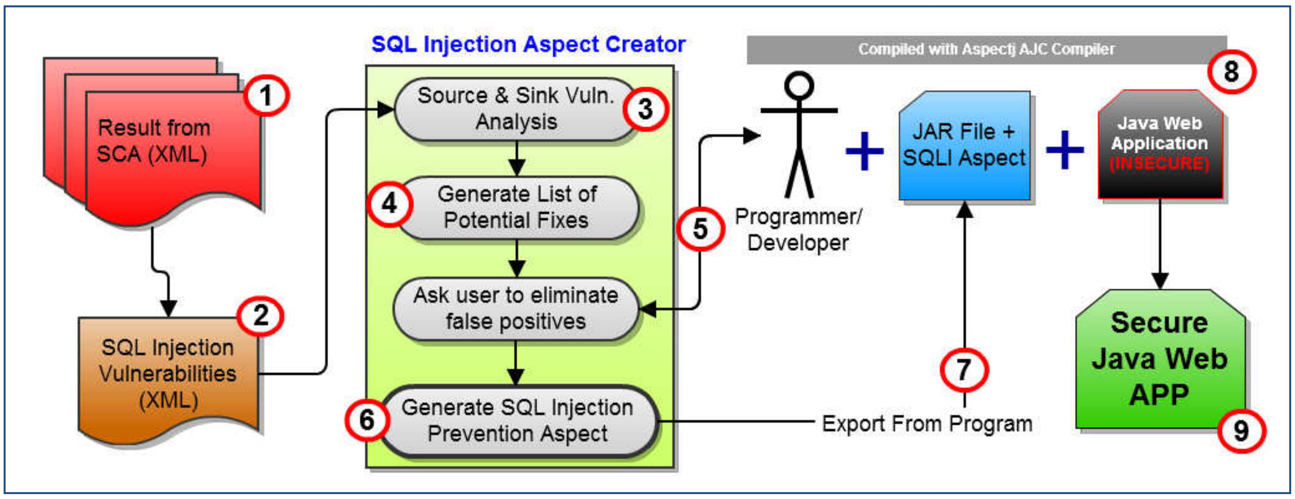
OWASP provides several rules for preventing XSS that are necessary to prevent scripts from being executed or stored into an application. These rules include items such as not allowing insertion of untrusted data except in specific locations, escaping HTML input, escaping attributes, escaping JavaScript, and escaping of several other types of possible input. The ESAPI encoding library is also used for doing the various types of encoding. Since it is impossible for the program to determine what types of input are to be inserted at each point, the developer executing the program created in this thesis will have to specify the type of encoding to be done. This approach was chosen over writing custom encoders because the ESAPI library has a comprehensive set of encoders for validating any type of input the user can provide. The XSS mitigation technique of white list validation is also implemented in this thesis. The user is able to choose from a list of default white list templates such as alphanumeric but they also have the ability to apply their own custom regular expressions. The AOP implementation is able to support the ESAPI encoding and validation libraries as well as the ability to apply custom regular expressions at each join point identified.

2.5 SQL Injection Mitigation Approach

There are several ways to approach fixing SQL Injection vulnerabilities in a web application. Since the vulnerabilities given by Fortify SCA are part of the legacy code of an application, it's important to come up with a method of fixing them without the need to modify it. AOP was chosen for this task because the paradigm can be used to write pointcuts that advise upon the exact file and line number given by the SCA. The approach can be summarized by the following steps:

1. Use the Fortify SCA result to export the report in a XML format. Exporting the result in this format will allow the program to parse the XML file easily and load the different vulnerabilities into Java Beans.
2. Isolate the SQL Injection vulnerabilities from all others in the exported report and identify the source and sink locations of the individual vulnerabilities. This step will also load the issues into individual Java objects and put into a list of SQLI vulnerabilities
3. Analyze the source and sinks for each issue and then specify which functions pointcuts will need to be required for. It will be necessary to determine how many parameters the function has and which parameters will need to be intercepted as well as the data type of the parameters.
4. The program will need to be able to implement different types of fixes for each issue. This can include encoding or rejection of potentially harmful data as well as rewriting of potentially harmful parts of the database query at runtime.
5. In order to eliminate false positives and prevent unnecessary writing of pointcuts, the user will be asked if they want to fix individual SQLI vulnerabilities detected and what kind of fix they need the program to implement.
6. Once the user provides their input, the program will determine what vulnerabilities to create pointcuts for and will make sure that each pointcut is only executed in the location that the user has specified. Locations are identified by the source file name and the issues individual line number within that file.
7. The user will then be given a Java Archive File (JAR) that will contain the necessary libraries to fix their vulnerabilities as well as the corresponding SQL Injection Prevention aspect.
8. The user will need to compile their application using the AspectJ compiler, AJC with the given JAR file and SQL Injection Prevention aspect by putting them in the build path of their application and incorporating the files into their build process.

9. Once the application is built, the user should be able to deploy their application as normal and the specified fixes for SQL Injection will take effect at runtime whenever the code identified by the SCA is executed.



SQL Injection Aspect Creator
Result from SCA (XML)
1
SQL Injection Vulnerabilities (XML)
2
Source & Sink Vuln. Analysis
3
Generate List of Potential Fixes
4
Ask user to eliminate false positives
5
Generate SQL Injection Prevention Aspect
6
Compiled with Aspectj AJC Compiler
Programmer/ Developer
JAR File + SQLI Aspect
Java Web Application (INSECURE)
8
7
Export From Program
Secure Java Web APP
9

Figure 3 - SQL Injection Aspect Creator

The generation of potential fixes (4) will be determined by The OWASP Enterprise Security API (ESAPI) libraries capabilities. Today, large banking corporations like American Express implement ESAPI in their efforts to achieve security and major government organizations such as the Space and Naval Warfare Systems Command use it as well [42]. The ESAPI is typically used to encode or validate user input that could be identified as SQL Injection.

In step five the user is asked to eliminate false positives. The program will provide them with a list of potential fixes for each SQL Injection issue detected by the SCA. The user will then provide an input that will correspond to a specific fix for that vulnerability. It is important to note that the user can select not to choose a fix which will eliminate that particular issue as a candidate for the pointcut implementation in the aspect code.

Some of the challenges and limitations of the above approach are imposed by the specific encoding libraries used for the data and the fact that they only support encoding for various types of SQL dialects such as Oracle and MySQL. Another important limitation is the lack of being able to detect an injected tautology. It soon became necessary to create a tautology detector for SQL queries because the attacker could inject tautologies such as “2 > 1” which would be evaluated to true. The aspect will need to detect such simple tautologies with different types of operators in mind and prevent them from being executed. One of the requirements for a successful implementation of the SQL Injection aspect and generated JAR file will be to compile the application using the AspectJ compiler. This may require the user to make modifications to their applications build script but will not require any changes to existing source code.

2.6 XSS Mitigation Approach

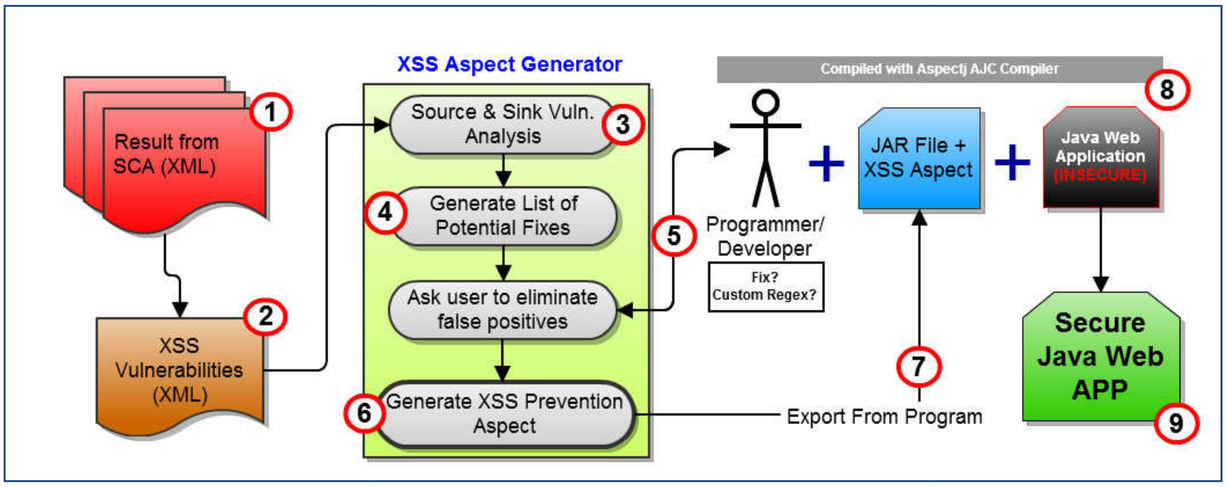
The Cross-site Scripting (XSS) approach will be similar in concept to the SQL Injection approach but will differ quite a bit in implementation. The XSS approach will implement a whitelist functionality that the user will have control over. The program will have to handle three main types of XSS threats that are detected by the Fortify SCA. The approach for mitigating XSS in Java web applications using AspectJ is outlined in the steps below:

1. Use the Fortify SCA result to export the report as an XML data file. This file will contain all the SQLI and XSS vulnerabilities of all types together in one location. The program will need the data in this particular format in order to parse it correctly.
2. The program will need to isolate the XSS vulnerabilities. It will need to make sure that it exports all three major types; persistent, reflected, and poor validation that is detected by the SCA. This will be done by parsing the particular “Category” nodes of the XML file and matching the value with the vulnerability name.
3. Next the identified XSS vulnerabilities will need to be stored into objects and the individual sources and sinks will need to be identified. Each source and sink will typically be associated with a method call within the source code. These function calls will need to be identified and stored into an XML file where they can later be referenced. Some of the information that will need to be captured in this step for each function is the number of parameters the method has and the data type of

each of these parameters. This is done so that the pointcut can later be created using this data.

4. The program will then need to generate a list of potential fixes based on the vulnerability. For XSS, this will be a list of whitelist options that the user will later have to choose from. These options are mostly provided by the ESAPI library mentioned above along with some of the more common whitelist regular expressions for web applications.
5. In this step the user will be asked to eliminate false positives. The user will be required to provide their input on whether or not they want each issue fixed and if so, what kind of validation they require to be done. If the user is not satisfied with the list of whitelist options, they should be able to provide their own regular expression that will then need to be implemented.
6. Once the users options are captured, an aspect will need to be created that will apply all the validation desired. With the validation implementation being in an aspect, the user will not need to modify their source code in order to have the fix take effect. In the creation of the aspect and its pointcuts, the execution of each one at runtime will be timed and logged using a logging functionality. This will ensure that the user has complete record of each piece of validation being executed.
7. Once the aspect is created, it will need to be exported so that the user can implement it into their insecure application. The program will export several other files necessary for the fixing of XSS vulnerabilities as well. The JAR file provided to the user will contain the necessary ESAPI libraries, logging libraries, and classes necessary for executing the validation. The XSS prevention aspect that is generated will also be provided outside of the JAR file.
8. Using the JAR file and aspect generated the user will need to include them in their project. The JAR file will typically be included into the lib of the project and incorporated into the build path in the users' application. The XSS prevention aspect will be put into the package structure of the application and the user will need to build the project with the AspectJ compiler.

9. When the application is successfully built, the user will need to deploy and execute their application as usual. The application will then have all of the desired validation implemented at runtime because the XSS prevention aspect's advice code will be weaved into the classes specified by the Fortify SCA in the first step.



XSS Aspect Generator
Compiled with Aspectj AJC Compiler
Result from SCA (XML)
1
XSS Vulnerabilities (XML)
2
Source & Sink Vuln. Analysis
3
Generate List of Potential Fixes
4
Ask user to eliminate false positives
5
Programmer/ Developer
Fix? Custom Regex?
Generate XSS Prevention Aspect
6
Export From Program
7
JAR File + XSS Aspect
Java Web Application (INSECURE)
8
Secure Java Web APP
9

Figure 4 - Creation of the XSS Mitigation Aspect

In step (2) the program will need to parse each Issue node of the XML and look for each of the three different types of XSS vulnerabilities. The source and sinks will need to be also identified manually by analyzing the different types of functions that are reported to have XSS vulnerabilities. Each of these functions will be identified and their attributes will be stored into the XML file for later reference by the program. Figure 5 shows the data that will need to be parsed and put into individual objects in the code as specified by bullet (3) above.

In step 5 for the XSS prevention aspect creation the process differs from the SQLI aspect approach in a couple of ways. First, the program will provide the user with a different set of solutions that are specified by the ESAPI for XSS prevention. The user will also have the feature of providing their own whitelist regular expression if they feel the default list provided for them is incomplete. This will ensure the least amount of false positives because the user, presumably someone with extensive experience with the application, will have control over what parts of their application they need to implement security into.

The next several steps will require the application developers' effort. The developer will need to use the AspectJ compiler in order to do the compile time weaving required for the aspects to be woven into the application and the vulnerabilities to be fixed. The application will provide the developer with the proper documentation that will cover the download and installation of the compiler and all software that is required for these steps. Since the AspectJ compiler (AJC) is just an extension of the Java language, the user will not need to have any special version of the JDK installed in order for the aspects to work as intended. The AOP code will seamlessly work with the users existing Java version.

2.7 Determining Success Criteria

The success criteria of this project will vary between the efforts to mitigate SQL Injection and XSS. This section of the chapter will be organized into three sections. The first will be the success criteria for SQL Injection, the second will be for XSS and the third will be the cost of fixing the vulnerabilities with respect to performance.

2.7.1 Success Criteria for SQL Injection

When determining success criteria for SQL Injection mitigation, it's most important to keep in mind the application performance and the assurance that the application will continue to function without additional error. In order to achieve this, several critical tests must be done. First, the application's logic that does the SQLI mitigation will need to be extensively unit tested at the method level. Second, all three applications used for the case study will need to be tested. This testing of the applications will be manual and since it would be nearly impossible to test each pointcut's execution manually because of the size of the application, the pointcuts will only be spot tested. In the testing of the pointcuts, the result of each request will be compared to the normal execution of the application and if there are any exceptions, the pointcut will be considered a failure. Third, each piece of code executed in the effort to mitigate SQLI will be timed. The elapsed time to fix each of the vulnerabilities will then be analyzed and any code that is found to be too expensive to execute will be omitted.

2.7.2 Success Criteria for XSS

In the effort to mitigate XSS, the XSS Preventions Aspect biggest success criteria will be to validate the applications input and outputs affected by the pointcuts without affecting normal execution flow. Although each part of the validation will be timed and analyzed, the length of time to whitelist a user's is typically negligible because it is simple string manipulation. The challenge to maintain normal execution flow comes from the ESAPI libraries that are used to do the whitelisting and validation. These libraries typically throw varying exceptions when the validation fails so it will be critical to handle these as they arise. The XSS prevention aspect will also be spot tested in all three enterprise applications used for this research. Since the user has the ability to enter a custom regular expression for the whitelisting functionality of the aspect, the application will have to gracefully handle regular expressions in incorrect format.

2.7.3 Success Criteria for Performance

The performance success criterion for the XSS and SQLI prevention aspects is perhaps the most important to keep in mind when applying security. Since the AOP paradigm is weaving code that fixes vulnerabilities into the applications legacy code, it will be important to not affect the users experience when using the application. In order to ensure this, the application will have to make sure that no particular path of execution for the vulnerabilities exceeds an allotted amount of time. Every possible path of execution will need to be timed and analyzed by the JUnit 4 test suite for the Java programming language. These tests will verify that the application performs successfully various tests in a reasonable amount of time. Tests will be containing both normal input parameters as well as possibly malicious ones and will come from various sources including OWASP.

These JUnit tests will be created for both the XSS and SQLI aspects and will contain scripts to run every possible combination of attacks and mitigation algorithms that the aspects would possibly execute in a live application environment. Once the aspects are deployed as part of an application, each execution of individual pointcuts will be timed and logged using Log4j and made available to the user for analysis. If the user is ever unsatisfied with the performance of the AOP implementation, they would simply have to redeploy their application without the aspects and corresponding libraries.

Chapter 3 Implementation

This chapter will go into details regarding the process of successfully creating and implementing SQL Injection and XSS prevention aspects in Java web applications. The aspects will be created and tested for the Alfresco, JadaSite, and OFBiz applications chosen as case studies for this paper. The remainder of this chapter will be organized as follows. The first section will describe the effort of choosing these three applications and static analysis efforts of each one. The second section will go into detail of analyzing the results of the Fortify SCA and using them to create the different SQL Injection and XSS Mitigation aspects. The third section will contain the implementation of algorithms to mitigate SQLI and XSS vulnerabilities. The fourth section will give a description of the deployment process for the aspects and the aspect generator’s user interface.

3.1 Identifying Case Studies and Static Analysis

This section will discuss the case studies chosen to act as validation for the papers’ success. Each of the projects were retrieved from various open source libraries available on the Internet and specific reasons for choosing them will be discussed below. The section will also explain the process that is required to analyze the source code for each application and will provide a thorough explanation on the steps are required to achieve a consistent and valid result.

3.1.1 Criteria for Applications to Analyze

In order to successfully implement this program, it soon became obvious that a major success factor would be to provide examples of how it could be applied in a real world scenario. We have chosen to use three open source applications as case studies to prove the program is valid and applicable across different industries. In order to find these projects, websites such as FreshMeat.net, SourceForge.net, and GitHub.com were searched since they contain large amounts of open source Java based projects. In the search process there were several criteria that needed to be met in order to make sure that the projects were compatible with the chosen AspectJ AOP implementation.

Some of the criteria for choosing the projects include the fact that it has to be a Java program because of the AOP implementation, AspectJ. Since the main focus of this implementation is preventing XSS and SQLI in web applications we limited the search to applications with web based architectures. It was also important to choose applications that had enterprise level implementations in the real world because it would potentially show how insecure these current implementations might be and also solidify that the efforts of this paper are necessary. Another important aspect in choosing the case studies comes from the frequency of change submissions and general activity of each application's user community. Choosing applications that have an active community meant that if the source code analyzer was able to find vulnerabilities, and the program was able to mitigate vulnerabilities, then the successful AOP implementation would be much more significant since vulnerability mitigation would be proven on active projects.

In the process of choosing case studies it was important to choose applications that fit within the range of technologies that are available to the project. One of these restrictions was the decision to use the ESAPI from OWASP in order to do whitelisting and encoding capabilities. For this reason, applications with the database implementations that did not support MySQL or Oracle were excluded. This is because the ESAPI encoder library supports only two types of codecs when doing SQL encoding. Although the AOP implementation created by this project does not limit the functionality for individual application servers, for testing purposes it was necessary to choose only projects that supported deployment with an Apache Tomcat application server. This was done because the Tomcat server is open source and it supports multiple versions of the Java programming language and could be easily deployed on a Windows operating system where the testing was done. Another technology that was chosen solely for the purpose of simplifying testability was the choice to only choose Java applications built on top of Java 5 or Java 6. This would make the testing of each of the projects easier because it did not require several different installations of Java on the same machine used for testing.

One of the final criteria that had to be fulfilled was the general size of the application. Applications with less than three hundred class files were omitted from consideration because they were typically not robust enough to be considered enterprise level implementations. The process for doing this type of elimination consisted of searching through the various applications' source repositories and validating the size of their source code directories. As each of the criterions was considered, the list of candidate applications dwindled.

3.1.2 Identifying Applications for Case Studies

With all of the restrictions applied from the criterion above, the list of applications wasn't too long as they were applied to all three online databases searched. One of the first results considered was the Alfresco Open Source Content Management (ECM). This project provides document management, collaboration, records management, knowledge management, web content management, and imaging for businesses. Alfresco makes use of some of the latest Java technologies such as the Spring MVC framework, Hibernate Object Relational Mapping (ORM) framework, and advanced search indexes such as Lucene. Alfresco was chosen because it has several enterprise level implementations in the real world. Companies such as Endeca, Yell.com, Orbitz, and Office Depot all make use of this applications powerful ECM solution [43]. The framework is a leading open source alternative for enterprise content management and it has over 140 thousand community members, over two-thousand enterprise customers, and over three million downloads. Since Alfresco is generally a content management system, this meant that this application typically has many opportunities for users to submit custom input. This was one of the reasons that this application was chosen since each source of users' input could potentially be a SQLI or XSS attack. The application also supports the MySQL database that made it compatible with the ESAPI encoder libraries. The Alfresco application also has a very active community. The users' forums are very active with very recent change requests and feature suggestions. The applications' developers were also proven to be active with regular code submissions with frequencies of at least once a month.

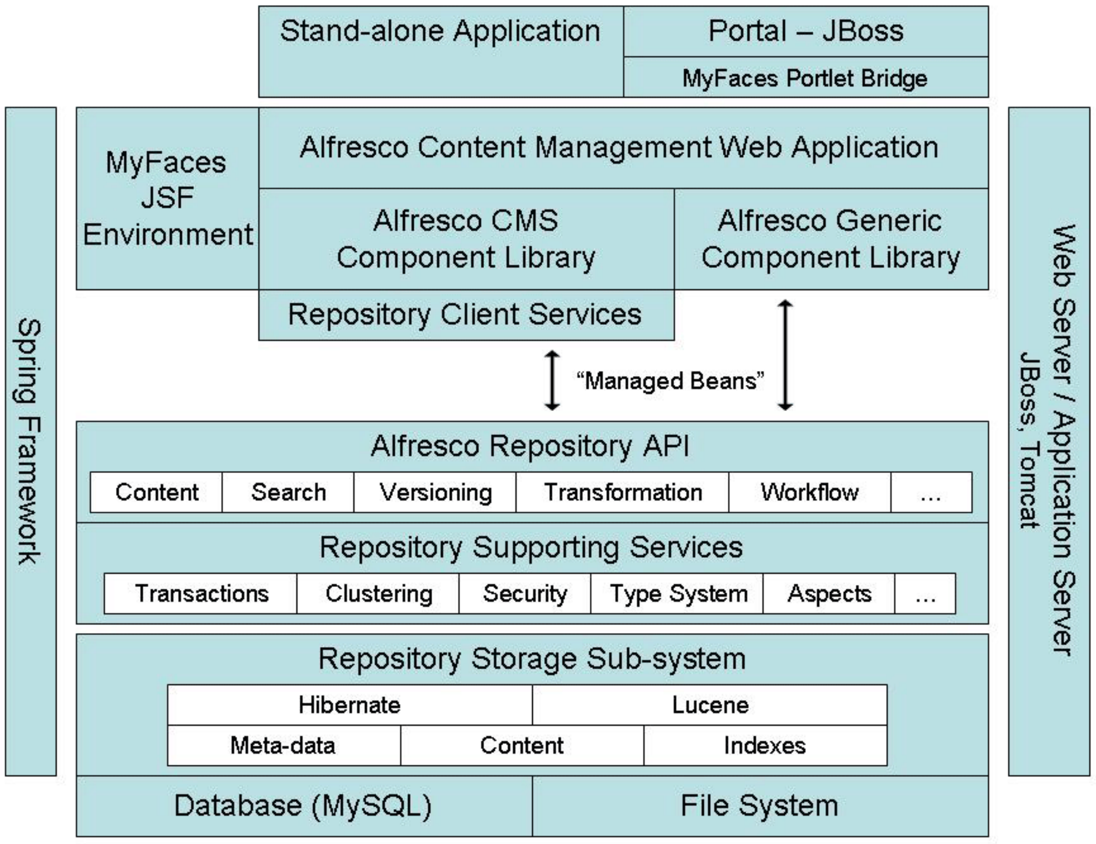


Figure 5 - Alfresco Architecture

The second open source project chosen was Apache's OfBiz. This application that is under the Apache open source umbrella is considered one of the best Enterprise Resource Planning (ERP) implementations available today. The OfBiz project is database agnostic and will function with any database that has a JDBC driver available. This agnostic approach to the data layer means the data is organized in an Object Oriented fashion with a relational database concept in mind. The presentation layer is typically composed of Java Server Pages (JSPs), and Widgets which are unique to this framework. The business layer of the application is composed of thousands of Java classes that are organized in a Model View Controller (MVC) framework. The project also makes use of several other technologies such as Apache Ant which would become important when compiling the application using the AspectJ compiler. The fact that this project implements a non-traditional framework will also show how the power of AOP can be leveraged to work with projects that tend to break the mold of traditional Java frameworks.

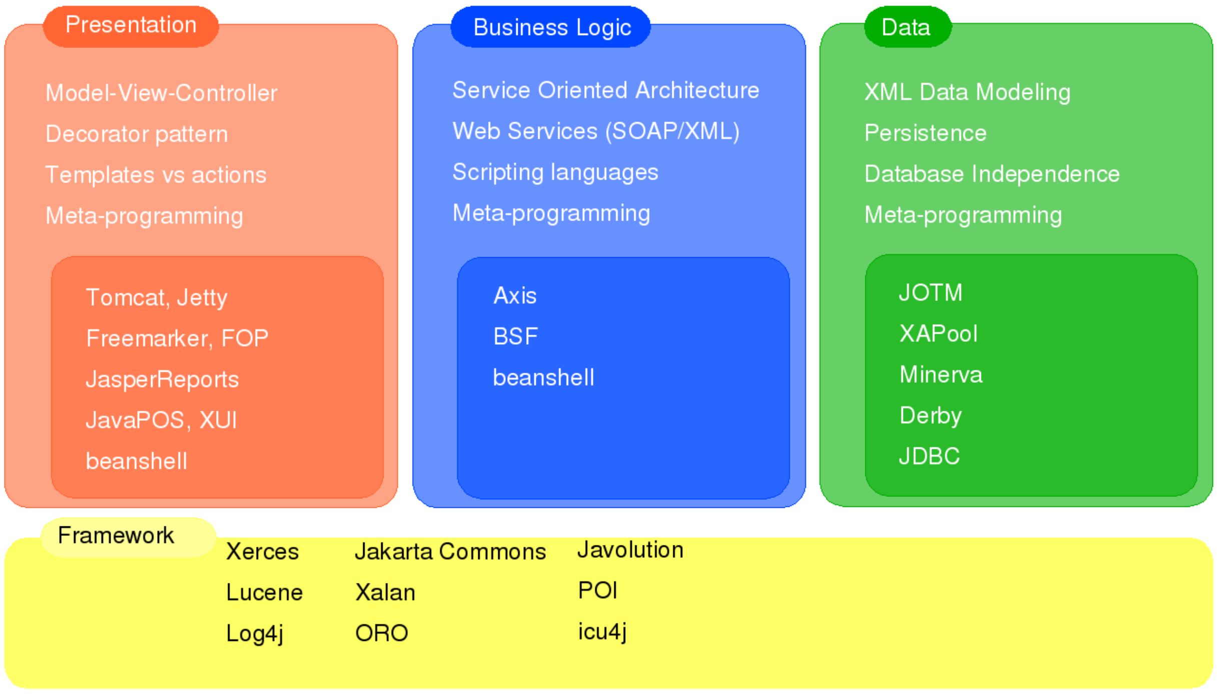


Figure 6 - OfBiz Architecture

The OfBiz project also has some other aspects that made it appealing for analysis. The project has a very active community with new code being submitted for either enhancements or bug fixes on a daily basis. The projects' customers also include some enterprise level implementations such as 1800flowers.com (E-Commerce), DKNY (E-Commerce), and BT.com (UKs largest ISP). The appeal towards OfBiz extends to the fact that E-Commerce applications, the most common implementation of this framework, are often the victim of many XSS and SQLI attacks because it collects PII such as credit card information and addresses from thousands of users every day.

The third open source project chosen for this paper was JadaSite. This application is another implementation with an E-Commerce capability similar to that of OfBiz. The main factor that differentiates this project from the others is its support for international storefronts and character sets. This application also makes extensive use of multiple web 2.0 technologies such as AJAX and WYSIWYG user interfaces. The

JadaSite project makes use of a MySQL database and uses JSPs as the primary source for generating the view for the user.

In the search for enterprise level implementations of Java web applications; these three selections seemed to be most suitable for the task at hand considering the criteria that needed to be enforced from the previous section. The projects selected are thought to be sufficient in ensuring success for a program that successfully mitigates SQLI and XSS vulnerabilities. The fact that these three applications have different implementations using similar technologies gives confidence in the fact that a solution that successfully mitigates these attacks for these projects can be applied to others as well.

3.1.3 Using the Fortify SCA

The ability to detect vulnerabilities in the three web applications chosen for case studies is the first step in creating a program that mitigates SQLI and XSS. Some of the benefits of using the Fortify SCA include [44]:

- Ensuring that software is trustworthy – The tools' technology identifies more vulnerabilities than any other detection method.
- Prioritizes Security Vulnerabilities – Provides information to the user on which issues need to be resolved first.
- Correlated Results – Correlates results across static and dynamic analysis to lower the chance of critical software vulnerabilities not being detected.
- Provide Best Practices – Provide the user with guidance on how to fix individual vulnerabilities detected by the SCA.

In order to use the Fortify SCA, it was necessary to have it installed and have the scan server running. The program was installed at the NKU facilities and the hosting server could be accessed remotely. Before the source analysis can be initiated on a code repository, each of the projects had to be copied onto the same server as the SCA. The three projects selected all had either subversion or CVS remote repositories that were available to check code out of. The version chosen from the remote repositories is the latest stable version of each application.

```
# Checking out JadaSite
cvs -d :pserver:anonymous@jadasite.cvs.sourceforge.net:/cvsroot/jadasite jadasite
cvs co -P jadasite

# Checking out Apache OfBiz
svn co http://svn.apache.org/repos/asf/ofbiz/trunk ofbiz

# Checking out Alfresco
svn co http://svn.alfresco.com/repos/alfresco-open-mirror/alfresco/HEAD
```

Figure 7 - Checking out Case Study Projects

Once the projects were checked out, they had to be prepared for the source code analyzer. In order for the analysis to execute successfully, there could not be any build path issues such as duplicate definitions of class files. To prevent such issues, each projects corresponding .classpath file had to be considered and all classpath entries that were specified to be excluded had to be removed from consideration before executing the SCA on each of the projects. Once the classpath and build paths of each project were determined, the scan could be executed as illustrated in the figure below:

```
# Running Fortify SCA for Apache OfBiz
sourceanalyzer -64 -Xmx2G -b ofbiz -clean
sourceanalyzer -64 -Xmx2G -source 1.5 -b ofbiz -cp \
        "Thesis/ofbiz/**/*.jar" \
        "Thesis/ofbiz/framework/**/*.java"
sourceanalyzer -64 -Xmx2G -source 1.5 -b ofbiz -scan ofbiz.fpr
```

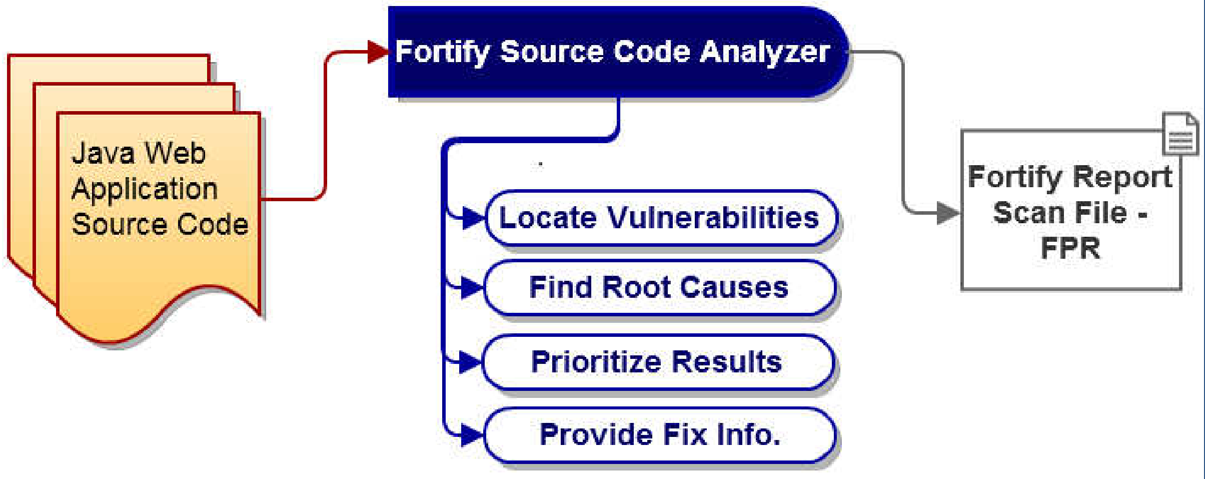

Java Web Application Source Code
Fortify Source Code Analyzer
Locate Vulnerabilities
Find Root Causes
Prioritize Results
Provide Fix Info.
Fortify Report Scan File - FPR

Figure 8 - Executing the Fortify SCA

When the source code analyzer is executed, the first step is called the translation phase. This phase uses the sourceanalyzer command to translate all of the libraries necessary to compile the project and its source code. Since the translation phase is memory intensive, it was necessary to provide a flag for the command to have 2GB of heap space available. Since the server was on a 64 bit OS, the -64 flag was necessary to execute the command under the 64 bit JRE. The –b parameter of the command corresponds to the build ID for the particular project that would later be referenced by the scan step of the analysis process. Other attributes such as the Java version the application uses, necessary libraries, and source code files to execute needed to be specified as well.

In the Analysis/Scan phase of the process, the intermediate file generated by the translation phase is scanned and the analysis results file (FPR) is created. The sourceanalyzer command is executed with the –scan, -b, and –f attributes. The –b attribute represents the build ID specified in the translation phase, the –scan attribute is used to tell the source analyzer to execute the scan phase, and the –f attribute specifies the resulting FPR file name.

3.1.4 Using the Fortify Audit Workbench

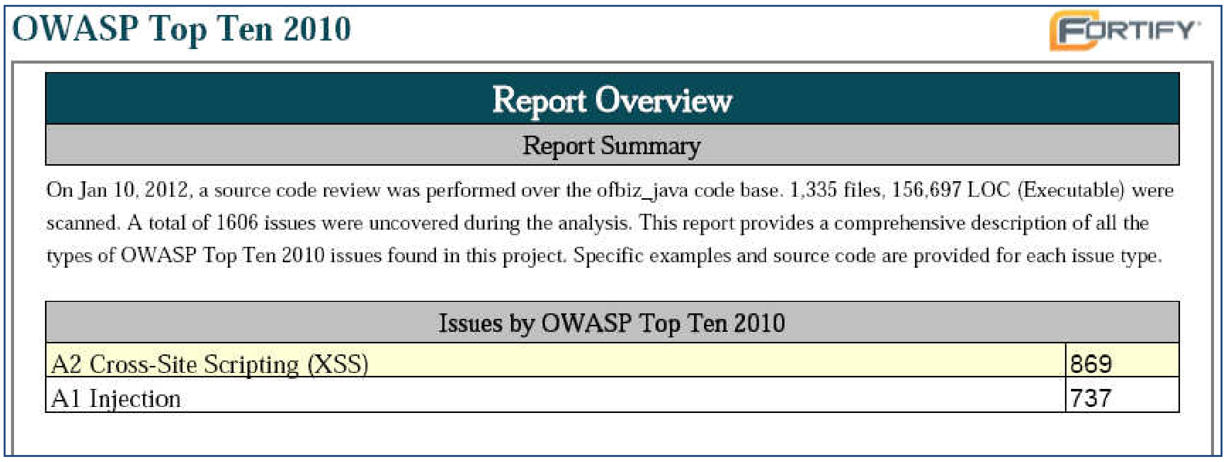
The Fortify Audit Workbench (AWB) product allows users to analyze the result given by the source code analyzer. The workbench has the ability to drill into specific issues to make auditing of issues easier. One of the biggest strengths of the AWB is the user interface design and the report generation features. Reports can be generated in several different formats such as .xml, .doc, .html, .pdf, and more. The report generation feature can also export the data in several methods. The method used for this paper was the “OWASP Top 10 – 2010” report which exports these top web application security threats this thesis covers [45]:

1. A1 – Injection

   Injection flaws, such as SQL, OS, and LDAP injection, occur when untrusted data is sent to an interpreter as part of a command or query. The attacker’s hostile data can trick the interpreter into executing unintended commands or accessing unauthorized data.

2. A2 – Cross Site Scripting (XSS)
   XSS flaws occur whenever an application takes untrusted data and sends it to a web browser without proper validation and escaping. XSS allows attackers to execute scripts in the victim's browser which can hijack user sessions, deface web sites, or redirect the user to malicious sites.

Since this paper only focuses on SQLI and XSS vulnerabilities, it was necessary to only export those two categories which are the top two according to OWASP to minimize the file size. The next step was to export reports for all three FPR files generated. The three files, jada_java.fpr, ofbiz_java.fpr, and alfresco_java.fpr were opened using the AWB and reports were generated in both XML and PDF formats for analysis. The results were exported and are shown in the figures below:



OWASP Top Ten 2010

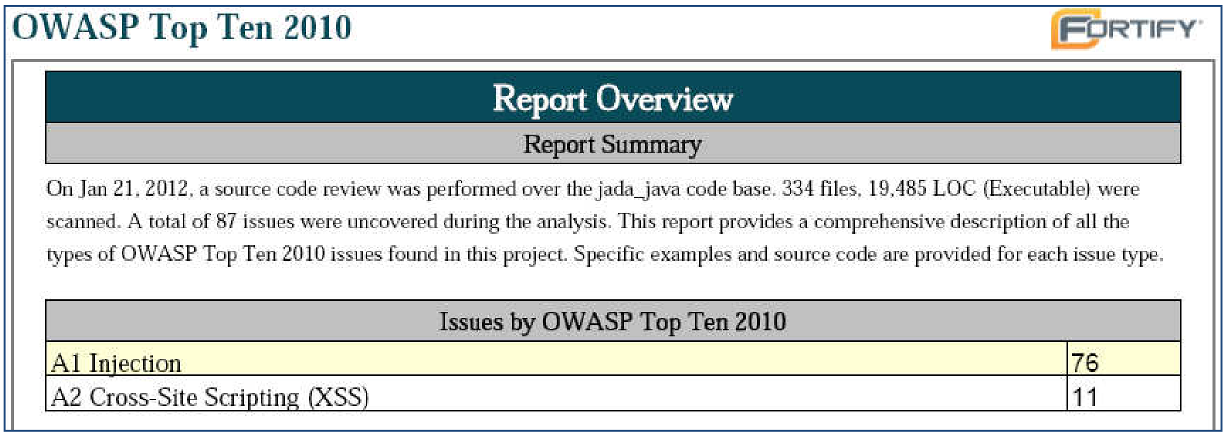
Report Overview

Report Summary

On Jan 10, 2012, a source code review was performed over the ofbiz_java code base. 1,335 files, 156,697 LOC (Executable) were scanned. A total of 1606 issues were uncovered during the analysis. This report provides a comprehensive description of all the types of OWASP Top Ten 2010 issues found in this project. Specific examples and source code are provided for each issue type.

Issues by OWASP Top Ten 2010	
A2 Cross-Site Scripting (XSS)	869
A1 Injection	737

Figure 9 - Apache OfBiz Report Results


OWASP Top Ten 2010

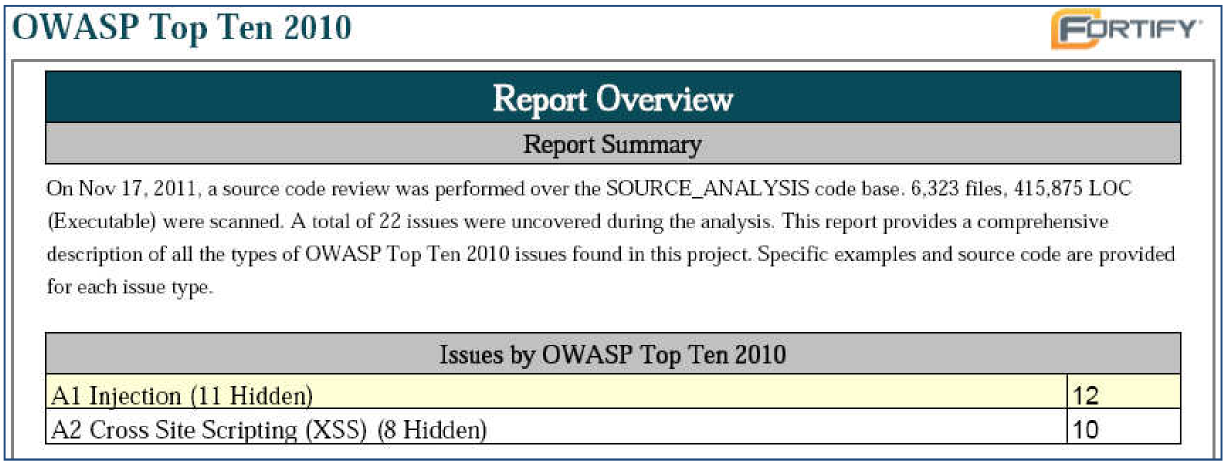
Report Overview

Report Summary

On Jan 21, 2012, a source code review was performed over the jada_java code base. 334 files, 19,485 LOC (Executable) were scanned. A total of 87 issues were uncovered during the analysis. This report provides a comprehensive description of all the types of OWASP Top Ten 2010 issues found in this project. Specific examples and source code are provided for each issue type.

Issues by OWASP Top Ten 2010	
A1 Injection	76
A2 Cross-Site Scripting (XSS)	11

Figure 10 - JadaSite Report Results


OWASP Top Ten 2010

Report Overview

Report Summary

On Nov 17, 2011, a source code review was performed over the SOURCE_ANALYSIS code base. 6,323 files, 415,875 LOC (Executable) were scanned. A total of 22 issues were uncovered during the analysis. This report provides a comprehensive description of all the types of OWASP Top Ten 2010 issues found in this project. Specific examples and source code are provided for each issue type.

Issues by OWASP Top Ten 2010	
A1 Injection (11 Hidden)	12
A2 Cross Site Scripting (XSS) (8 Hidden)	10

Figure 11 - Alfresco Report Results

With the reports all generated in XML format as well, the task of using the AWB was completed and the tool did not require any further use. The next step was to use the generated files and write a program to analyze them.

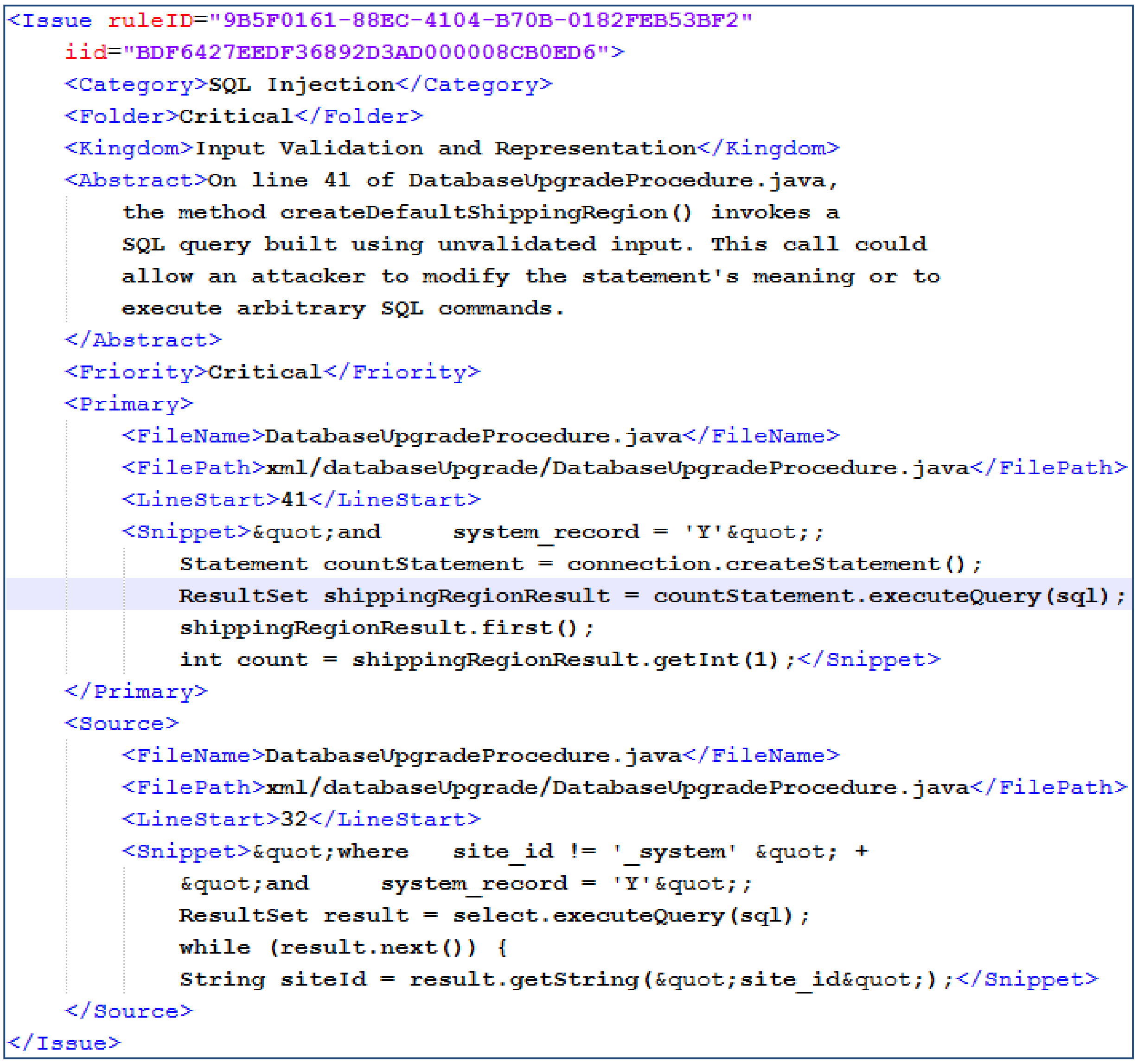
3.2 Fortify SCA Result Analysis

This section will go into detail on the effort to analyze the results of the source code analyzer and the audit workbench. The second part of this section will discuss the process required to identify the pointcuts that will need to be created as part of the finished product. The next two sections will discuss the template file creation for the pointcuts and advice pieces of each aspect. These template files will be used by the program to generate the aspects necessary for mitigating SQLI and XSS vulnerabilities.

3.2.1 Parsing the XSS and SQL Injection Issues

The first step towards parsing the XML file which contains all of the SQLI and XSS vulnerabilities is to look at the anatomy of it. As shown in the figure below, each issue has the following attributes:

- Issue – contains identifier values for each of the individual issues.
- Category – specific category for that issue
- Folder – the priority and importance attribute for the issue.
- Kingdom – the domain that the particular issue belongs to.
- Abstract – detailed description of the vulnerability with explanation on how it can be exploited.
- Friority – priority category given by the Fortify tool.
- Primary – this section contains the information for the vulnerability with respect to its implementation in the source code.
- Filename – the file the vulnerability is located in from the source code analyzed.
- Filepath – Location of the source file with respect to the projects main directory.
- Linestart – The exact line number the vulnerability occurs inside of the filename specified in the same "Primary" tag.
- Snippet – The piece of code retrieved from the source file where the vulnerability occurs.
- Source – this tag contains the same attributes as the Primary tag. The data referred in this tag corresponds to the sink of the vulnerability.



```
<Issue ruleID="9B5F0161-88EC-4104-B70B-0182FEB53BF2"
    iid="BDF6427EEDF36892D3AD000008CB0ED6">
    <Category>SQL Injection</Category>
    <Folder>Critical</Folder>
    <Kingdom>Input Validation and Representation</Kingdom>
    <Abstract>On line 41 of DatabaseUpgradeProcedure.java,
        the method createDefaultShippingRegion() invokes a
        SQL query built using unvalidated input. This call could
        allow an attacker to modify the statement's meaning or to
        execute arbitrary SQL commands.
    </Abstract>
    <Friority>Critical</Friority>
    <Primary>
        <FileName>DatabaseUpgradeProcedure.java</FileName>
        <FilePath>xml/databaseUpgrade/DatabaseUpgradeProcedure.java</FilePath>
        <LineStart>41</LineStart>
        <Snippet>&quot;and     system_record = 'Y'&quot;;
            Statement countStatement = connection.createStatement();
            ResultSet shippingRegionResult = countStatement.executeQuery(sql);
            shippingRegionResult.first();
            int count = shippingRegionResult.getInt(1);</Snippet>
    </Primary>
    <Source>
        <FileName>DatabaseUpgradeProcedure.java</FileName>
        <FilePath>xml/databaseUpgrade/DatabaseUpgradeProcedure.java</FilePath>
        <LineStart>32</LineStart>
        <Snippet>&quot;where   site_id != '_system' &quot; +
            &quot;and     system_record = 'Y'&quot;;
            ResultSet result = select.executeQuery(sql);
            while (result.next()) {
            String siteId = result.getString(&quot;site_id&quot;);</Snippet>
    </Source>
</Issue>
```

Figure 12 - Anatomy of Issue from SCA Result

With the anatomy of the issue tag in the SCA result identified, the next step was to parse each of these issue nodes and load them into Java objects. The Issue class was created for this purpose. Each of the source and sinks (i.e. Primary and Source tags) were treated as separate issues in order to provide full coverage of vulnerabilities. The Issue class is a plain java object that contained attributes that would be relevant to creating the final aspects. The object had two constructors, one with no parameters and one that accepted all attributes as parameters.

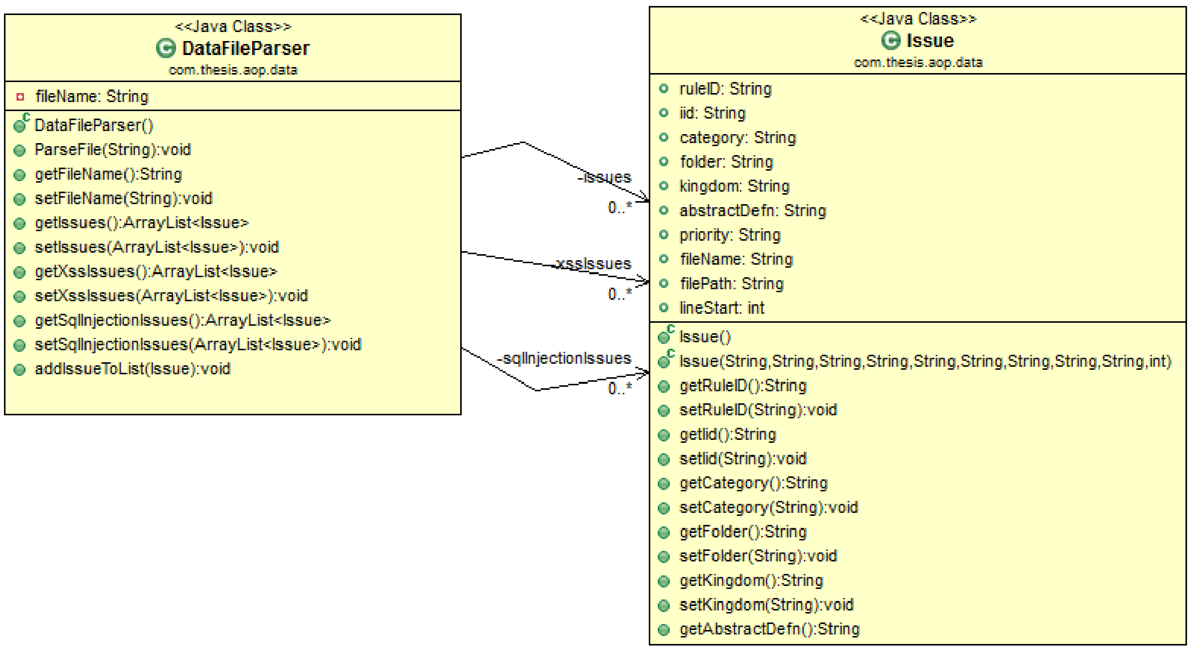


Figure 13 - Class Diagram DataFileParser and Issue

Now that a container object was created for holding the individual issues, it was necessary to create a class that did parsing of the file. This class called DataFileParser accepted the filename of the data file to be parsed. The class makes use of the SAXParser XML parsing library. When an instance of the SAXParser is given a data file, it uses the content handler (in this case, the DefaultHandler) object to iterate through the files individual nodes. As the parser iterates through each element of the XML file, whenever it encounters the “ISSUE” or “SOURCE” element in the startElement method, a new Issue object is created. Once the parser is inside one of these elements, it iterates through the inner elements and these are then captured into the corresponding properties of the current Issue object inside the “characters” method of the DefaultHandler. When the closing tag for each issue is encountered in the endElement method, the current Issue is stored into an ArrayList that will contain all of the Issue objects created as part of the parsers execution. One important piece to mention is that the “SOURCE” elements do not have their own abstract definition, category, folder, kingdom, or priority. This is because this node is always a child node of the “ISSUE” element and therefore it will inherit these attributes because they would be the same.

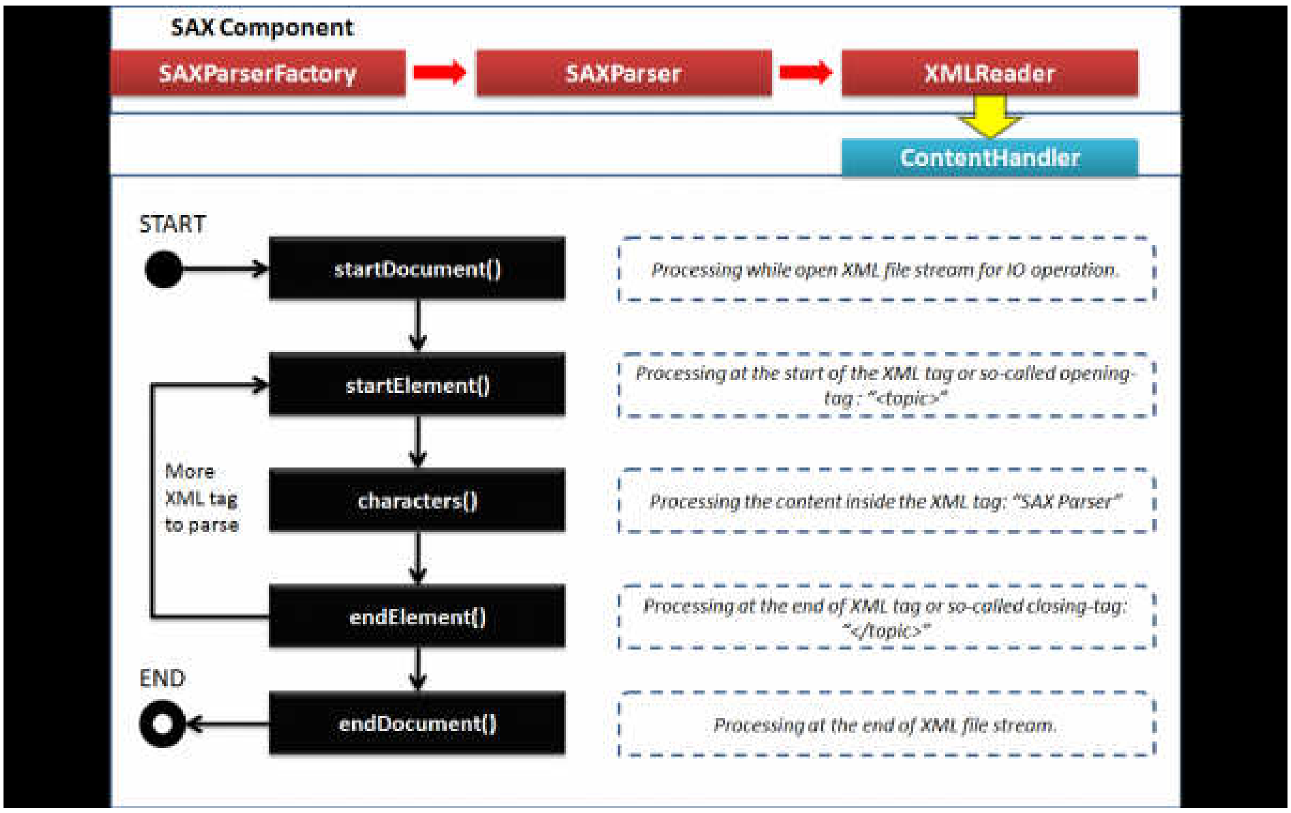

SAX Component
SAXParserFactory
SAXParser
XMLReader
ContentHandler
START
startDocument()
startElement()
characters()
endElement()
endDocument()
More XML tag to parse
END
Processing while open XML file stream for IO operation.
Processing at the start of the XML tag or so-called opening-tag : "<topic>"
Processing the content inside the XML tag: "SAX Parser"
Processing at the end of XML tag or so-called closing-tag: "</topic>"
Processing at the end of XML file stream.

Figure 14 - How SAX Parser Works [46]

Once all the "ISSUE" nodes are captured into the ArrayList, the next step is to separate the XSS and SQL Injection vulnerabilities. In order to achieve this, two more array lists of type Issue will need to be created that will hold the two different types of vulnerabilities. The main list will then be iterated and the category attribute of each Issue object will be checked. If the category is name contains the string "SQL Injection" or "Cross-Site Scripting", then that object is put into the corresponding array list. Once the two lists containing the two types of vulnerabilities are created inside the DataFileParser object, they can then be retrieved whenever they are necessary by the rest of the program.

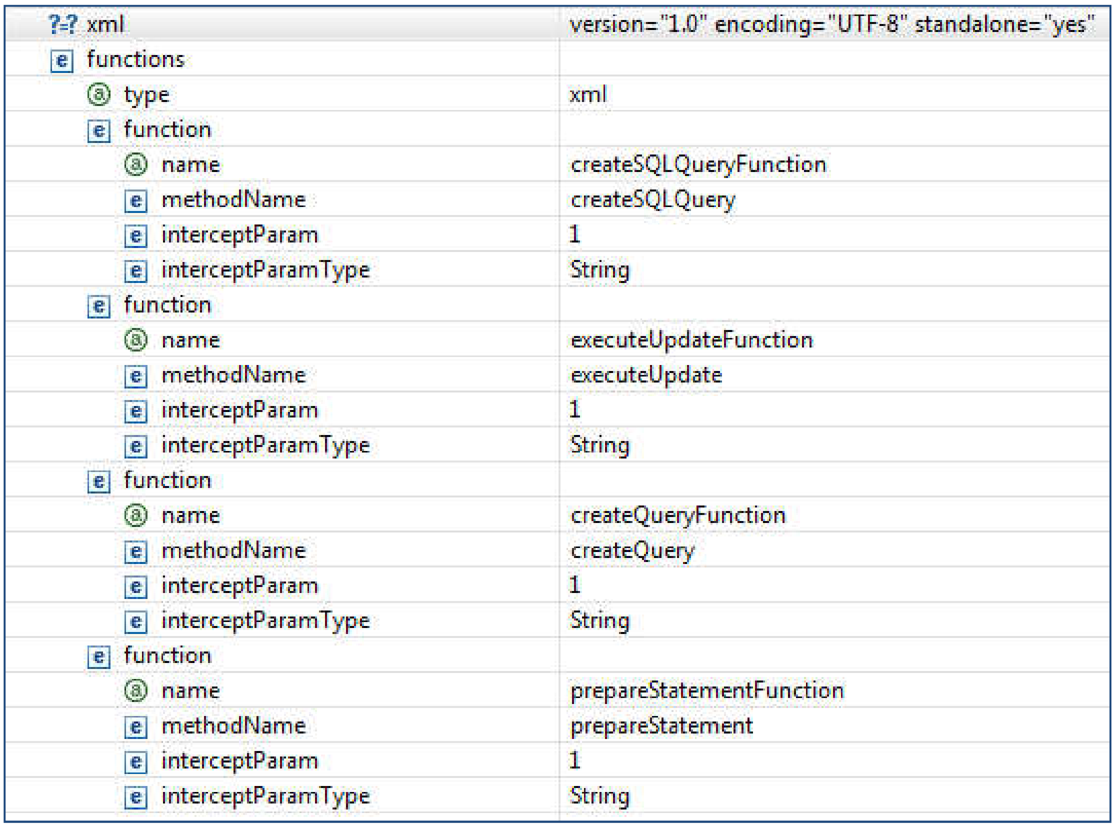
3.2.2 Identifying Pointcuts

After all of the vulnerabilities have been sorted into individual array lists where they can be easily accessed by the program, the next step was to identify the pointcuts the program would need to created. The approach taken to identify the pointcuts was to manually analyze all of the issues identified by the SCA and record the function calls that were the source of the vulnerability. This was done because it would be impossible to extract information such as data types of function parameters and the specific method return types. In the case of Cross-site scripting, these were often functions that either read the users input or potentially showed the user unvalidated data. For SQL Injection vulnerabilities, the vulnerabilities were usually located whenever a SQL query that contained unvalidated data was being executed. This list of potentially vulnerable functions included vulnerabilities inside of ORM frameworks such as Hibernate in some applications.

When these functions were identified, they had to be stored in an XML data file that could later be easily parsed and read using the above mentioned SAXParser. There were several pieces of information that needed to be captured for each of these functions in order to be able to create a successful pointcut:

- name – the name attribute of each function would contain the name of the pointcut.
- methodName – the name of the method the pointcut will be created for. This corresponds to the method in the source code identified by the SCA.
- interceptParam – the location of the parameter that needs to be intercepted by the pointcut. This parameter will contain the data that needs to be checked for potentially harmful scripts.
- interceptParamType – the data type of the parameter that needs to be intercepted. This was typically a String or byte array in the case of XSS and SQLI vulnerabilities.
- param – Each function node will consist of one or more param items. These represent the parameters that are being captured as part of the pointcut and advice implementations. Each param item will contain the type, name, location, and result attributes. The result attribute will be a Boolean that tells whether or not the current param value is the one to be intercepted at each join point.

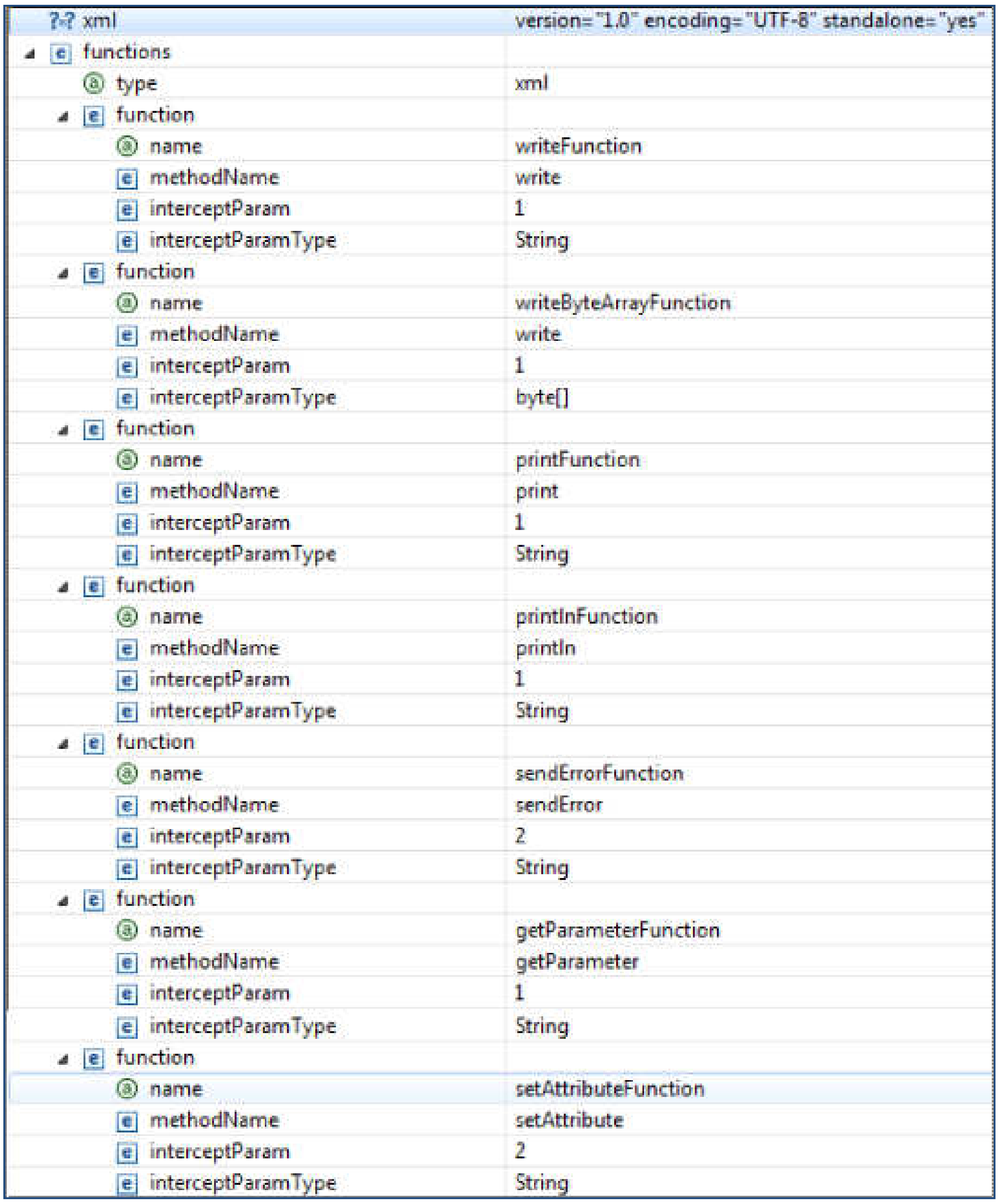
When the analysis was completed, the XSS functions captured for pointcuts were stored in an XML file named xssfunctions.xml and the SQLI functions were captured into a file named sqlInjectionFunctions.xml. The XSS functions file contained seven different function types that pointcuts would need to be created for and the SQL Injection functions file contained four. The functions are shown in the figures below.



?=? xml	version="1.0" encoding="UTF-8" standalone="yes"
e functions	
ⓐ type	xml
e function	
ⓐ name	createSQLQueryFunction
e methodName	createSQLQuery
e interceptParam	1
e interceptParamType	String
e function	
ⓐ name	executeUpdateFunction
e methodName	executeUpdate
e interceptParam	1
e interceptParamType	String
e function	
ⓐ name	createQueryFunction
e methodName	createQuery
e interceptParam	1
e interceptParamType	String
e function	
ⓐ name	prepareStatementFunction
e methodName	prepareStatement
e interceptParam	1
e interceptParamType	String

Figure 15 - SQL Injection Pointcut Functions

The vulnerabilities for SQL Injection were always associated with one of the above method executions across all three projects. In most instances where either of these methods was executed, the string parameter they took contained some sort of unvalidated data as determined by the source code analysis. This unvalidated data could allow an attacker to modify the SQL statement being executed or to execute its own queries that could cripple an application.



?-? xml	version="1.0" encoding="UTF-8" standalone="yes"
functions	
type	xml
function	
name	writeFunction
methodName	write
interceptParam	1
interceptParamType	String
function	
name	writeByteArrayFunction
methodName	write
interceptParam	1
interceptParamType	byte[]
function	
name	printFunction
methodName	print
interceptParam	1
interceptParamType	String
function	
name	printlnFunction
methodName	println
interceptParam	1
interceptParamType	String
function	
name	sendErrorFunction
methodName	sendError
interceptParam	2
interceptParamType	String
function	
name	getParameterFunction
methodName	getParameter
interceptParam	1
interceptParamType	String
function	
name	setAttributeFunction
methodName	setAttribute
interceptParam	2
interceptParamType	String

Figure 16 - XSS Pointcut Functions

The vulnerabilities identified by the source code analyzer across the three applications tested fit into the seven functions defined above. Some of these functions such as getParameter require pointcuts because they take data that could be harmful directly off of the request object without validating it. Other functions such as print and println, usually write out directly to a console or a log file. This can be a serious threat if the console logs or log files are ever compromised because this code could possibly be writing out PII.

Once the two pointcut function files are identified, they have to be parsed and stored into individual Function objects for which the Function class was created for. It is important to remember that multiple function nodes will need to be created for the same function if it has a different method signature. This class will contain attributes for each of the values in the function nodes from the XML files. The function parser class, FunctionsParser, will use the SAX parser and load the individual functions into objects. Once this is done, the parser will create array lists and put the SQL Injection and XSS functions into the correct list.

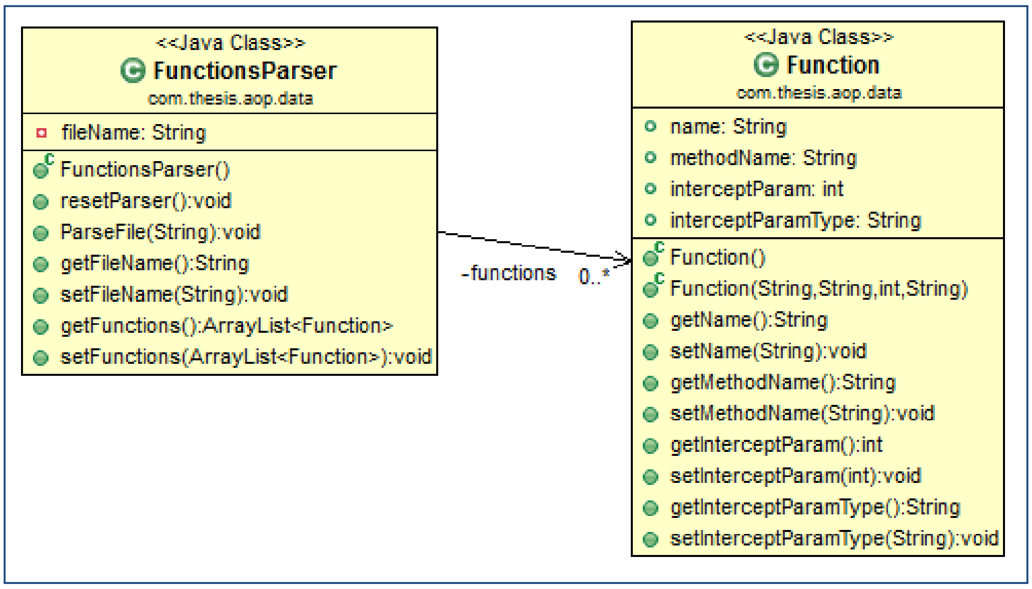


Figure 17 - Function Object and FunctionsParser Class Diagrams

3.2.3 Pointcut Template Creation

The next step in creating SQLI and XSS mitigation aspects was to define a template for pointcuts. A pointcut is a program element in AspectJ that identifies join points in the source code and exposes data from the execution context at those locations [47]. The pointcuts for this implementation make use of the pointcut designator and signature named “call” that accepts a parameter “MethodPattern”. Using this signature allows the pointcut to place a join point on each area where the method name matching the MethodPattern is being called. The SQL Injection pointcut template file is called sqlInjection.template and the XSS pointcut template file is called xss.template. Each of the template files would contain placeholders for code to be implemented by the aspect generator algorithms. The placeholders are surrounded by “<” and “>” characters in the template files and are defined below:

- pointcut_name – this placeholder will be the individual pointcuts name. The placeholder will be replaced by the function name from the pointcut function definition file. For the XSS pointucts, the pointcut name will be prepended by “xss_” and for the SQLI pointcuts, prepended by “sqlInjection_”.
- pointcut_param – this placeholder will contain the parameter string for the pointcut. This value will be determined by the interceptParamType attribute of the Function object. The data type will be followed by the parameter name. If the pointcut needs to intercept multiple parameters, they will be separated by commas.
- function_name – the functionName attribute from the Function object will be placed here because this is the function name from the source code that will need to be intercepted by the pointcut.
- function_params – this value will be replaced by the data types of the parameters the function to be intercepted accepts. This value will also come from the interceptParamType attribute from the Function object.
- within_string – The within string will contain the disjunction of all the file names that this pointcut will need to be applied to. These file names will be determined from by the Issue objects from the lists that contain the SQL Injection and XSS vulnerabilities.
- args_string – This value will contain the parameter names the pointcut gives to the values that are being intercepted.

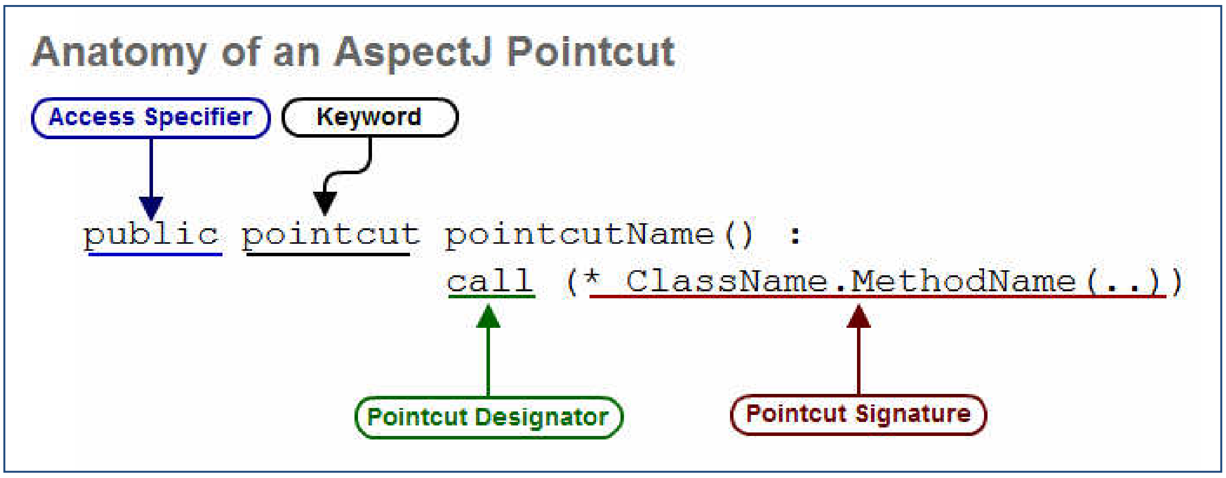

Anatomy of an AspectJ Pointcut
Access Specifier
Keyword
public pointcut pointcutName() :
call (* ClassName.MethodName(..))
Pointcut Designator
Pointcut Signature

Figure 18 - Anatomy of an AspectJ Pointcut

```
pointcut xss_<pointcut_name>(<pointcut_param>) :
    call(* *.<function_name>(<function_params>)) &&
    (<within_string>) &&
    args(<args_string>);
```

Figure 19 - XSS Pointcut Template

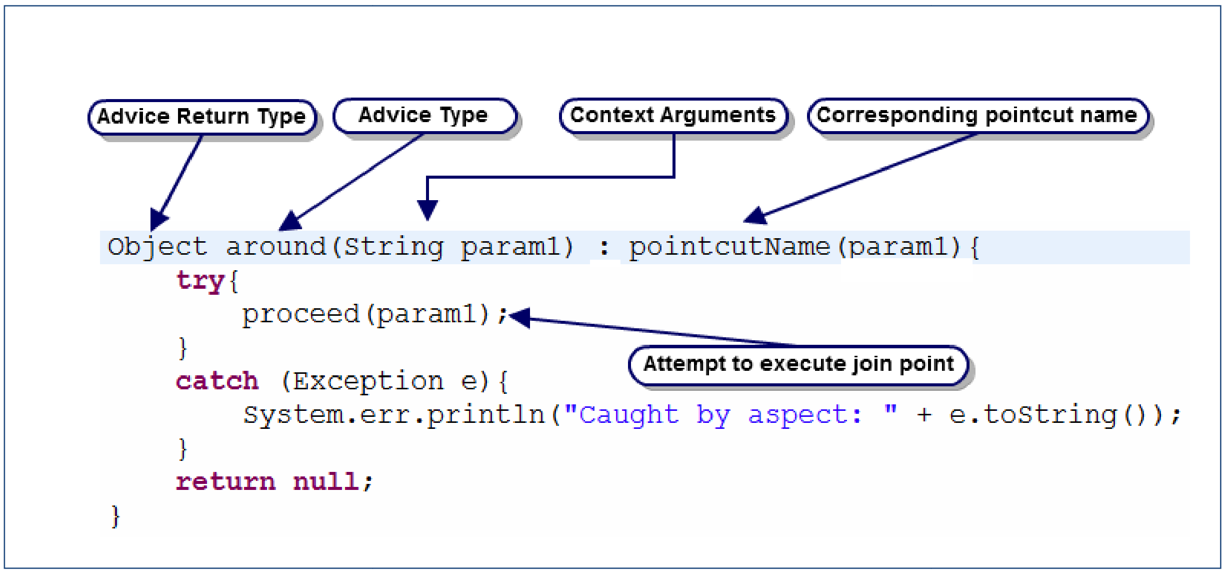
```
pointcut sqlInjection_<pointcut_name>(<pointcut_param>) :
    call(* *.<function_name>(<function_params>)) &&
    (<within_string>) &&
    args(<args_string>);
```

Figure 20 - SQL Injection Pointcut Template

With each of the pointcut templates created, the aspect generator will be able to use the placeholders defined in this section to populate each of them with code or data that will have a part in mitigating SQL Injection and XSS vulnerabilities. The template file approach to generating pointcuts provides modularity to the program by taking the source code generation out of the main program.

3.2.4 Advice Template Creation

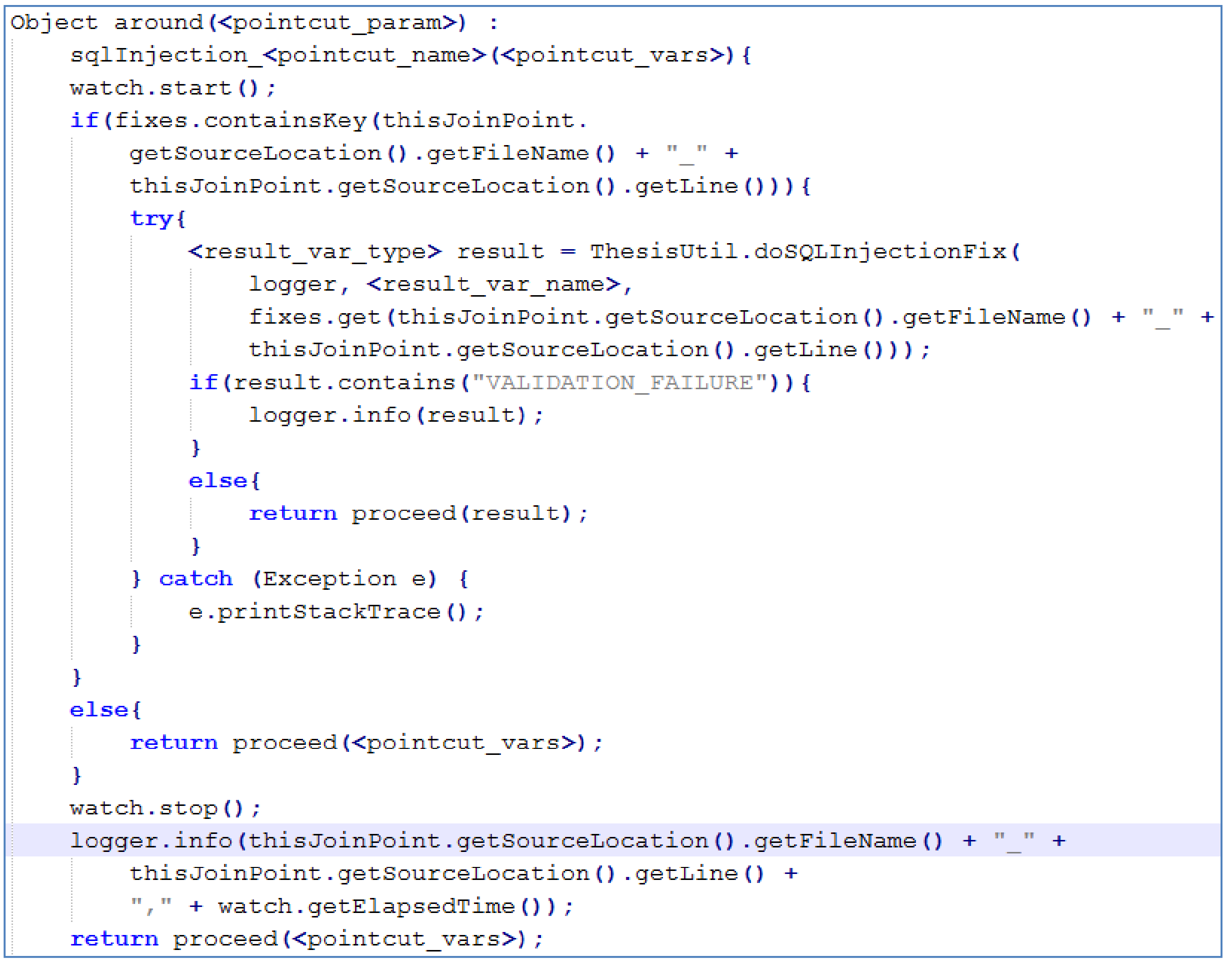
After creating the pointcut template files, the crosscutting behavior that will be executed at each join point needs to be defined. In AspectJ, this behavior is defined by the Advice section of the aspect. The advice will run at every join point that the pointcut picks matches in the application's source code. However, the code that is executed and how it is executed depends on the particular advice. The AspectJ AOP implementation supports three different types of advice. The after advice, has three different interpretations: After the execution of a join point executed normally, after it throws an exception, or after it does either one [48]. The before advice is simpler in the sense that it is only executed before the join point code. The around advice runs in place of the join point it operates over and since it can have a return value, it must be given a return type. Inside of the around advice, the original join point can be executed using the proceed function. This function takes the same arguments as the pointcut and returns whatever the advice is declared to return in an Object, even if it is a primitive value. The rest of this section will discuss the effort to create the template files for advices that will be used to mitigate SQLI and XSS vulnerabilities.



```
Object around(String param1) : pointcutName(param1){
    try{
        proceed(param1);
    }
    catch (Exception e){
        System.err.println("Caught by aspect: " + e.toString());
    }
    return null;
}
```

Figure 21 - Anatomy of Around Advice for AspectJ

After creating template files for the pointcuts that will create join points for the two types of vulnerabilities the SCA identifies, an advice will also need to be created for each pointcut. All of the advices will have a common template that will be applied when creating the final aspects. The advice templates will be part of the pointcut template files sqlInjection.template and xss.template. The aspects will implement an around advice that will be used to intercept the parameter that the join point would be executing by default. The advice template will be illustrated by the figure below:



```
Object around(<pointcut_param>) :
    sqlInjection_<pointcut_name>(<pointcut_vars>){
    watch.start();
    if(fixes.containsKey(thisJoinPoint.
        getSourceLocation().getFileName() + "_" +
        thisJoinPoint.getSourceLocation().getLine())){
        try{
            <result_var_type> result = ThesisUtil.doSQLInjectionFix(
                logger, <result_var_name>,
                fixes.get(thisJoinPoint.getSourceLocation().getFileName() + "_" +
                thisJoinPoint.getSourceLocation().getLine()));
            if(result.contains("VALIDATION_FAILURE")){
                logger.info(result);
            }
            else{
                return proceed(result);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
    else{
        return proceed(<pointcut_vars>);
    }
    watch.stop();
    logger.info(thisJoinPoint.getSourceLocation().getFileName() + "_" +
        thisJoinPoint.getSourceLocation().getLine() +
        "," + watch.getElapsedTime());
    return proceed(<pointcut_vars>);
```

Figure 22 - SQL Injection Aspect Advice Template

pointcut_param – This will be the placeholder for the context arguments mentioned in the figure above. This parameter will be the intercepted value that will need to be verified.

1. pointcut_name and pointcut_vars – The name of the pointcut whose join points will be advised on and the name of the parameters defined in the pointcut definition that it accepts.
2. The first if-statement checks the fixes map to see if the user has elected to do a fix at that file and line number that the join point is on. The fixes map will be described in the aspect generation section. The thisJoinPoint variable contains reflective information about the current join point being executed and can only be used in the context of the advice [49]. The variable is being used to get the exact

file name and line number where the join point occurs by using the source locations' .getFileName() and .getLine() methods.

3. The result_var_type and result_var_name placeholders will be replaced by the desired result variable type and name. For example, if the join point's method is expecting a byte[] instead of a String, the correct value will be replaced from the functions' XML definition. For this implementation, the only two types of values that will be evaluated are byte array and String objects. However, the value can easily be updated to support other return types for both the XSS and SQLI validators.
4. If the join point is in the fixes map, then the aspect will make a call to the doSQLInjectionFix method in the ThesisUtil class. The method will take the logger object, the unvalidated input the SCA defined as a potential threat, and the type of fix the user specified. The resulting string will either be a secure value or contain a validation failure.
5. In the case that the SQL fix encounters an error, the resulting string will contain the value "VALIDATION_FAILURE" at the beginning of it followed by a more detailed description such as a stack trace. The logger will then write out this error to the specified log file where the user will be able to analyze the output.
6. If the result of the doSQLInjectionFix method does not contain an error, the advice will call the proceed method that will execute the join points code with the new, secure value. It is important to remember that if the function is not able to fix the issue but did not encounter an error, it will return the same value that it was given. The result_string placeholder will be replaced by the value of the result parameter. In the event that the join point method takes multiple parameters, the correct value will be replaced by the result.
7. In the event that the pointcut defines a join point that the user chose not to implement a fix for, the code will execute as normal by using the proceed function and passing it the parameter values it would have received in its normal execution flow.
8. Once the advice has done its core logic, the logger will write out to a log file specifying the location of the join point the advice just executed for and the

duration of the execution. This will provide the user with an exact execution time for each join point that the aspect affects.

The Cross-site scripting prevention aspects advice template is a bit simpler then the SQL Injection one in the sense that it does not have the chance to return a result with the “VALIDATION_FAILURE” message. The advice also calls a different method, doXSSFix() to do the XSS vulnerability mitigation.

```
Object around(<pointcut_param>) : xss_<pointcut_name>
    (<pointcut_vars>){
    watch.start();
    if(fixes.containsKey(thisJoinPoint.getSourceLocation().
    getFileName() + "_" + thisJoinPoint.getSourceLocation().
    getLine())){
        try{
            <result_var_type> result = ThesisUtil.
            doXSSFix(<result_var_name>, fixes.get(thisJoinPoint.
            getSourceLocation().getFileName() + "_" +
                thisJoinPoint.getSourceLocation().getLine()));
            proceed(<result_string>);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
    else{
        proceed(<pointcut_vars>);
    }
    watch.stop();
    logger.info(thisJoinPoint.getSourceLocation()
    .getFileName() + "_" +
        thisJoinPoint.getSourceLocation().getLine() +
        "," + watch.getElapsedTime());

    return null;
}
```

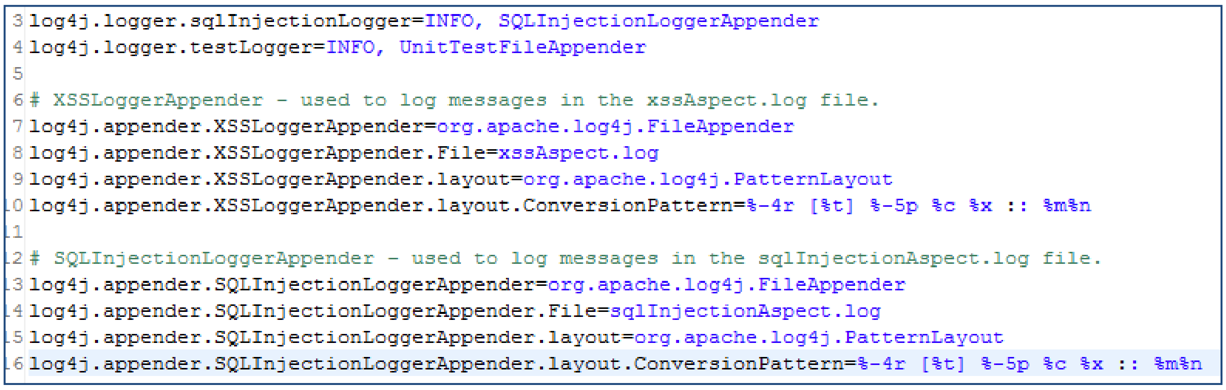
```
Object around(<pointcut_param>) :
    xss_<pointcut_name>(<pointcut_vars>){
    watch.start();
    if(fixes.containsKey(thisJoinPoint.getSourceLocation().
        getFileName() + "_" +
        thisJoinPoint.getSourceLocation().getLine())){

        try{
            <result_var_type> result = ThesisUtil.doXSSFix(<result_var_name>,
            fixes.get(thisJoinPoint.getSourceLocation().getFileName() + "_" +
                thisJoinPoint.getSourceLocation().getLine()));
            return proceed(<result_string>);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
    else{
        return proceed(<pointcut_vars>);
    }
    watch.stop();
    logger.info(thisJoinPoint.getSourceLocation().getFileName()
        + "_" + thisJoinPoint.getSourceLocation().getLine()
        + "," + watch.getElapsedTime());
    return proceed(<pointcut_vars>);
}
```

Figure 23 - XSS Aspect Advice Template

3.2.5 Aspect Logging

Whenever executing an aspect, it will be important to log every single execution of the join points created by the applications pointcuts in order to provide complete transparency to the application owners. This program will utilize the Apache Log4J logging library to track all executions at each join point. Log4j is a java based logging utility that offers logging at several different levels such as INFO, DEBUG, and ERROR. The logger can be configured via either properties or xml file. The properties file approach has an advantage over the XML approach because the logger settings can be manipulated by changing the file rather than source code. The FileAppender is one of the appenders available and is used for this implementation because all logging will be done to external files. The layout used will be the PatternLayout that uses a pattern string for outputting one line at a time.

There will be a logger created for each of the two aspects in the log4j.properties file. The logger will be instantiated in the constructor for each of the aspects created and the PropertyConfigurator object will be loaded with the log4j.properties file specified in the figure below. The log file names will be sqlInjectionAspect.log and xssAspect.log for the SQL Injection mitigation and XSS mitigation aspects, respectively.



```
log4j.logger.sqlInjectionLogger=INFO, SQLInjectionLoggerAppender
log4j.logger.testLogger=INFO, UnitTestFileAppender

# XSSLoggerAppender - used to log messages in the xssAspect.log file.
log4j.appender.XSSLoggerAppender=org.apache.log4j.FileAppender
log4j.appender.XSSLoggerAppender.File=xssAspect.log
log4j.appender.XSSLoggerAppender.layout=org.apache.log4j.PatternLayout
log4j.appender.XSSLoggerAppender.layout.ConversionPattern=%-4r [%t] %-5p %c %x :: %m%n

# SQLInjectionLoggerAppender - used to log messages in the sqlInjectionAspect.log file.
log4j.appender.SQLInjectionLoggerAppender=org.apache.log4j.FileAppender
log4j.appender.SQLInjectionLoggerAppender.File=sqlInjectionAspect.log
log4j.appender.SQLInjectionLoggerAppender.layout=org.apache.log4j.PatternLayout
log4j.appender.SQLInjectionLoggerAppender.layout.ConversionPattern=%-4r [%t] %-5p %c %x :: %m%n
```

Figure 24 - Log4j properties for the XSS and SQL Injection aspects

3.2.6 Generating the Aspects

With each of the pointcut and advice template files for both the Cross-site Scripting and SQL Injection prevention aspects created, the next task was to generate the aspects themselves. This process required several steps that will be described in this section. The section will be explaining the functionality of each Java class that was written to create both of the aspects and also go into detail on any specific libraries that needed to be leveraged. The three main classes that implement the aspect generation are called AspectGenerator, AspectBean, XSSAspectGenerator, and SQLInjectionAspectGenerator.

The AspectGenerator class (shown in diagram below) is the parent class of the XSSAspectGenerator and SQLInjectionAspectGenerator classes and contains several methods that are used to help generate the SQLI and XSS prevention aspects.

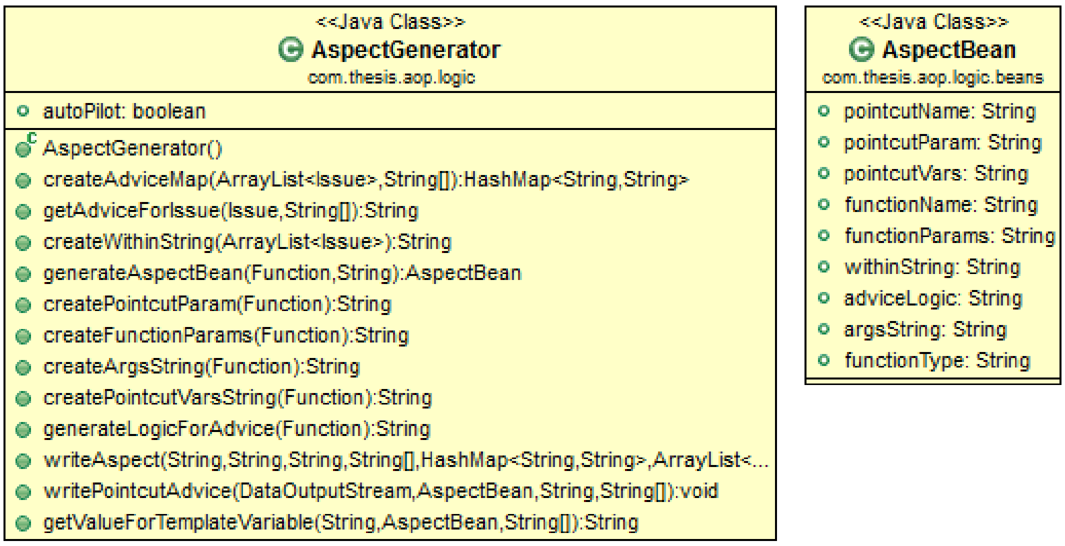

<<Java Class>>
AspectGenerator
com.thesis.aop.logic
autoPilot: boolean
AspectGenerator()
createAdviceMap(ArrayList<Issue>,String[]):HashMap<String,String>
getAdviceForIssue(Issue,String[]):String
createWithinString(ArrayList<Issue>):String
generateAspectBean(Function,String):AspectBean
createPointcutParam(Function):String
createFunctionParams(Function):String
createArgsString(Function):String
createPointcutVarsString(Function):String
generateLogicForAdvice(Function):String
writeAspect(String,String,String,String[],HashMap<String,String>,ArrayList<...
writePointcutAdvice(DataOutputStream,AspectBean,String,String[]):void
getValueForTemplateVariable(String,AspectBean,String[]):String
<<Java Class>>
AspectBean
com.thesis.aop.logic.beans
pointcutName: String
pointcutParam: String
pointcutVars: String
functionName: String
functionParams: String
withinString: String
adviceLogic: String
argsString: String
functionType: String

Figure 25 - Aspect Generator and Aspect Bean Class Diagrams

The table below outlines the methods in the AspectGenerator class, each parameter of the methods, and the overall functionality and implementation of each one.

createAdviceMap	ArrayList – Issues String[] Fix Options Returns - HashMap<String, String>	The createAdviceMap method will take the corresponding aspects issues (either XSS or SQLI) generated from the SCA and will call the getAdviceForIssue method for each of these issues. The map key will be the issue file name and the line number. The value portion of the map will be the fix that the user chose to implement for that particular issue.
getAdviceForIssue	Issue Object i String[] fixOptions Returns - String	The getAdviceForIssue method will take the issue object it is given and ask the user what kind of fix to implement based on the options in the fixOptions array. The user will also be provided with the issue category, ID, file name, file line number, priority, and the abstract from the Fortify SCA report. For testing purposes, the user can supply the digit “99” and the program will choose a random option from the fixOptions array.
createWithinString	ArrayList<Issue> issues	The createWithinString method will

	Returns - String	generate the within string to be used for the <within_string> placeholder in the pointcut template file. The method will iterate over the list of issues and create a disjunction of each filePath value of each one.
generateAspectBean	Function Object f String withinString Returns – AspectBean Object	The generateAspectBean method will create a new instance of the AspectBean class and set all of the necessary attributes. The AspectBean class will contain the data that will be used to populate all of the placeholders in the pointcut and advice templates identified in previous sections.
createPointcutParam	Function Object f Returns - String	This method will take a Function object and use its interceptParamType and interceptParam to create the parameter for the pointcut definition placeholder. This return value is the concatenation of the type and parameter number.
createFunctionParams	Function Object f Returns – String	This method will take the Function object and use its interceptParam and interceptParamType attributes to create the <function_param> placeholder replacement string. The method will also iterate through the individual "param" values of the function node and construct the string. The challenge in this method is to implement logic to create the string when the parameter being intercepted is not the first one the function accepts.
createArgsString	Function Object f Returns – String	This method uses the Function object and creates the parameter string that will be used to replace the <args_string> placeholder in the pointcut template file. The function objects param list will be iterated and the string will be constructed. The args_string value will be used to reference the parameter that the pointcut needs to intercept.
createPointcutVarsString	Function Object f Returns – String	This method is similar to createArgsString.
createResultString	Function Object f Returns – String	This method will iterate through the parameter list in the Function object and when it encounters the parameter

		that is specified as the result, it will replace its name with the “result” value that will represent the return value of the fix method implemented by each advice. This value will be used to replace the placeholder <result_string>
generateLogicForAdvice	Function Object f Returns – String	This method will use the method name on the function object and generate a comment used for separating out sections for each advice.
writeAspect	String - templateFileName String – resultFileName String – poinctutFileName String[] templateVariables HashMap<String, String> adviceMap ArrayList<AspectBean> aspectBeans	The writeAspectMethod will generate the aspect using the AspectBean objects passed to it. The template file for the aspect (shown in figure below) will first be opened and put into a DataInputStream. The file will be read line by line and the necessary placeholders will be replaced. The <insert_fix_map> placeholder will be replaced by the users’ suggested fixes and will be a set of generated put instructions to populate the aspects map that will contain all of the fixes. The <insert_code_here> placeholder will be replaced by the result of the writePointcutAdvice method for each of the AspectBeans provided to the method. The method will finish by closing out the resultFileName and also catch any exceptions that may occur.
writePointcutAdvice	DataOutputStream – aspectDataOutputStream AspectBean – AspectBean String – pointcutFileName String[] templateVariables FileInputStream templateInputStream	This method will open the pointcut and advice template file, iterate over each line, and apply replacement strings for each of the placeholders. The replacement strings will be provided by the getValueForTemplateVariable method.
getValueForTemplateVariable	String templateVariable AspectBean aspectBean String[] templateVariables Returns – String	This method will return the replacement string for the desired templateVariable. The template variable is the name of the placeholder in the pointcut and advice template file.

```
package com.aspects;
import com.thesis.aop.util.ThesisUtil;
import com.thesis.aop.util.StopWatch;
import org.apache.log4j.Level;
import org.apache.log4j.Logger;
import org.apache.log4j.PropertyConfigurator;
import java.util.HashMap;
import org.owasp.esapi.errors.IntrusionException;
import org.owasp.esapi.errors.ValidationException;
import net.sf.jsqlparser.JSQLParserException;


public aspect SQLInjectionAspect{
	public HashMap<String, String> fixes = new HashMap<String,
String>();
	Logger logger;
	StopWatch watch;

	public SQLInjectionAspect(){
		PropertyConfigurator.configure("log4j.properties");
		logger = Logger.getLogger("sqlInjectionLogger");
		logger.info("SQL Injection Aspect Created");
		watch = new StopWatch();

		<insert_fix_map>
	}

	<insert code here>
}
```

Figure 26 – SQL Injection Aspect Template

```
package com.aspects;
import com.thesis.aop.util.ThesisUtil;
import com.thesis.aop.util.StopWatch;
import org.apache.log4j.Level;
import org.apache.log4j.Logger;
import org.apache.log4j.PropertyConfigurator;
import java.util.HashMap;
import org.owasp.esapi.errors.IntrusionException;
import org.owasp.esapi.errors.ValidationException;
import net.sf.jsqlparser.JSQLParserException;


public aspect XSSAspect{
      public HashMap<String, String> fixes = new HashMap<String,
String>();
      Logger logger;
      StopWatch watch;

      public XSSAspect(){
            PropertyConfigurator.configure("log4j.properties");
            logger = Logger.getLogger("xssLogger");
            logger.info("XSS Aspect Created!");
            watch = new StopWatch();

            <insert_fix_map>
      }

      <insert_code_here>
}
```

Figure 27 - Cross-Site Scripting Aspect Template

The XSSAspectGenerator class uses the methods above along with some of its own to create the XSS mitigation aspect. The class has a constructor that takes an ArrayList of Function objects that are generated by the FunctionsParser class that utilizes the SAXParser to retrieve the functions defined in the xssfunctions.xml file and the list of XSS issues. These two parameters are used to execute the generateAspect method that creates the XSS Mitigation Aspect. The SQLInjectionAspectGenerator's generateAspect method has a similar functionality but instead it uses the array list of issues that contain the SQLI vulnerabilities, SQL Injection pointcut, and template file. Each of the methods is shown below to display the file names being passed to the writeAspect method.

```
	public void generateAspect() throws IOException {
		String withinString = createWithinString(xssIssues);

		adviceMap = createAdviceMap(xssIssues, xssFixOptions);
			for (Iterator iterator = functions.iterator();
			iterator.hasNext();) {
			Function f = (Function) iterator.next();
			aspectBeans.add(generateAspectBean(f, withinString));
		}
		System.out.println(aspectBeans.size());
		writeAspect("XSSAspect.java.template",
				"XSSAspect.aj",
				"xss.template",
				templateVariables,
				adviceMap,
				aspectBeans);
	}
```

Figure 28 - generateAspect method from XSSAspectGenerator class

```
public void generateAspect() throws IOException {
	String withinString = createWithinString(sqlInjectionIssues);

	adviceMap = createAdviceMap(sqlInjectionIssues,
	sqlInjectionFixOptions);

	for (Iterator iterator = functions.iterator();
	iterator.hasNext();) {
		Function f = (Function) iterator.next();
		aspectBeans.add(generateAspectBean(f, withinString));
	}
	System.out.println(aspectBeans.size());
	writeAspect("SQLInjectionAspect.java.template",
			"SQLInjectionAspect.aj",
			"sqlinjection.template",
			templateVariables,
			adviceMap,
			aspectBeans);
}
```

Figure 29 - generateAspect method from SQLInjectionAspectGenerator

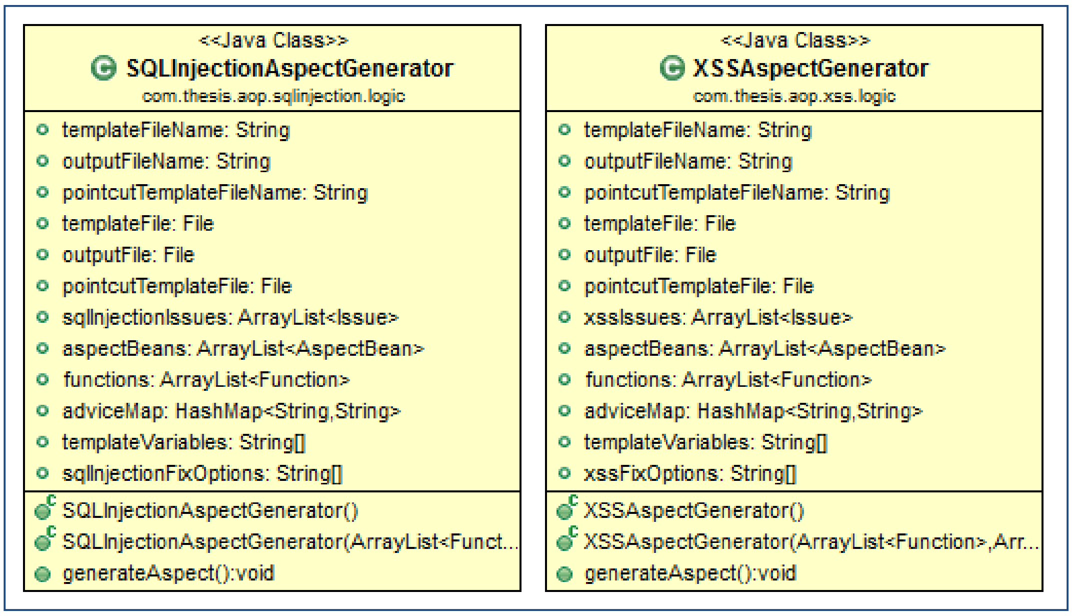

<<Java Class>>
SQLInjectionAspectGenerator
com.thesis.aop.sqlinjection.logic
templateFileName: String
outputFileName: String
pointcutTemplateFileName: String
templateFile: File
outputFile: File
pointcutTemplateFile: File
sqlInjectionIssues: ArrayList<Issue>
aspectBeans: ArrayList<AspectBean>
functions: ArrayList<Function>
adviceMap: HashMap<String,String>
templateVariables: String[]
sqlInjectionFixOptions: String[]
SQLInjectionAspectGenerator()
SQLInjectionAspectGenerator(ArrayList<Funct...
generateAspect():void
<<Java Class>>
XSSAspectGenerator
com.thesis.aop.xss.logic
templateFileName: String
outputFileName: String
pointcutTemplateFileName: String
templateFile: File
outputFile: File
pointcutTemplateFile: File
xssIssues: ArrayList<Issue>
aspectBeans: ArrayList<AspectBean>
functions: ArrayList<Function>
adviceMap: HashMap<String,String>
templateVariables: String[]
xssFixOptions: String[]
XSSAspectGenerator()
XSSAspectGenerator(ArrayList<Function>,Arr...
generateAspect():void

Figure 30 - SQLInjectionAspectGenerator and XSSAspectGenerator Class Diagrams

This section presented the functionality implemented to generate the XSS and SQL Injection prevention aspects. The generation of each aspect shares much of the same code which makes this project very extensible because all of the parameters can be modified and passed to the generator classes. If the user wanted to add additional pointcuts to be implemented, they would just need to add entries to the corresponding functions XML file. The program contains a main class that basically takes the XML result from the SCA and creates instances of the classes above to generate the two aspects.

3.3 Algorithms for Vulnerability Mitigation

This section will discuss the implementation of the logic that applies the SQL Injection and XSS mitigation. The SQL Injection and XSS mitigation aspects call the doXSSFix and doSQLInjectionFix methods to execute validation algorithms for a parameter that is intercepted via the pointcut at each join point that the user specified a fix for. The ThesisUtil class then takes the parameters passed in by the aspect and calls the getSolution method that will apply the desired fix on the input string and return a secure value. The getSolution method will map the desired fix value it receives to the list of possible fixes and call the necessary method for implementing that particular fix. If the method executed successfully, a valid string will be returned to the advice that was being executed. The remainder of this section will analyze the libraries and algorithms used to mitigate SQL Injection and XSS vulnerabilities and how they were implemented.

3.3.1 SQL Injection and ESAPI

When attempting to mitigate SQL Injection vulnerabilities there are typically three primary choices that programmers have. The first is to use parameterized queries or prepared statements. This method makes sure that an attacker is not able to modify the query that's being executed. For example, dynamically generated queries could potentially have a user input that contains a tautology such as "admin OR '1' = '1'" that would always result in the query returning a value. The use of prepared statements defines the queries' structure at compile time in order to prevent the attacker from changing its intent. A less effective solution is to use stored procedures. The stored procedure approach is similar to the prepared statements but the queries are stored in the database itself and then called by the application. In order to implement them on legacy applications it can be a significant level of effort. For example, implementing parameterized queries would require the rewriting of all modules that contain dynamically created queries and stored procedures would require the development team to move all their queries to the database layer.

The fact that implementing parameterized queries or stored procedures can be a significant undertaking to incorporate into a legacy application, the proposed approach is to escape all user supplied input before executing any query that could potentially contain malicious content. This is typically the less desired alternative since it can leave an application vulnerable to second level SQL injection attacks but it is the best one available if rewriting the query execution modules of an application is not an option. The implementation of this solution will be done by the SQL Injection mitigation aspect that will apply encoding of the users' input whenever one of the methods identified by the Fortify SCA is being executed. The library used to encode the input is the ESAPI encoder library. The ESAPI encoder library currently supports database encoding for the Oracle and MySQL databases with support for SQL Server and PostgreSQL being developed at this time. The ESAPI encoding implementation is illustrated by the figure below:

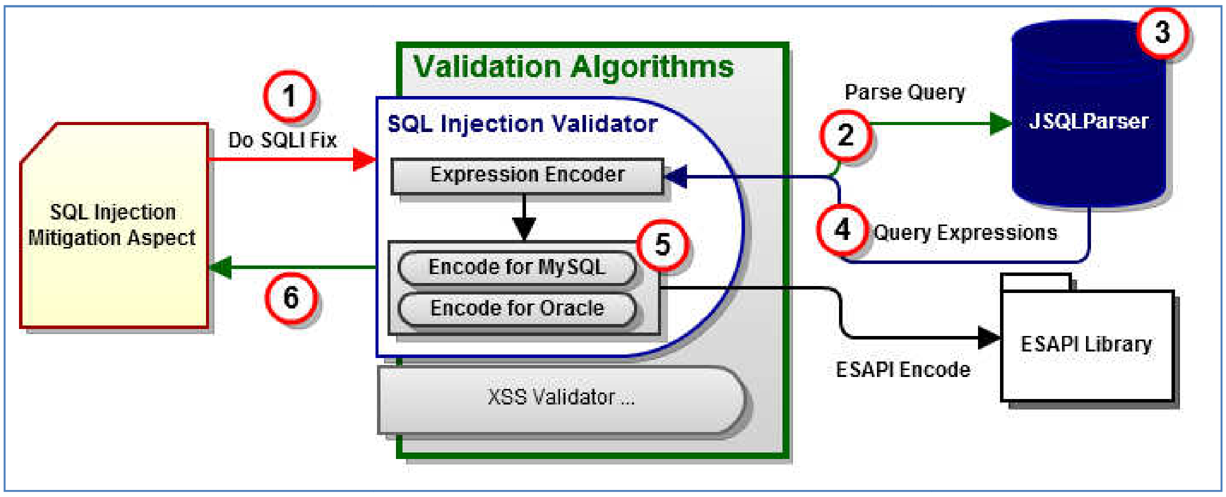
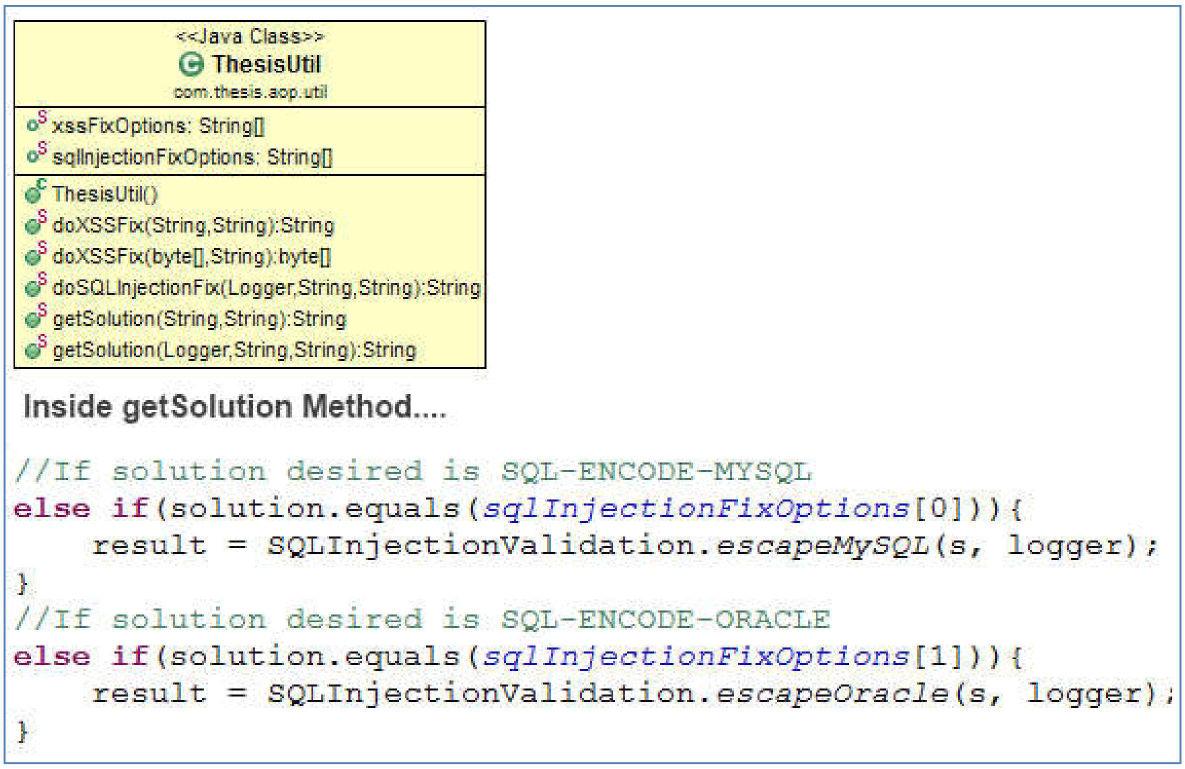


Figure 31 - SQL Injection Validator Execution Flow

1. In step one the SQL Injection mitigation aspect generated by the previous section will call its doSQLInjectionFix method in the ThesisUtil class. The method will then execute the getSolution method passing it the query that needs to be validated and the users' desired fix which would either be to encode the query for MySQL or Oracle.


```
<<Java Class>>
ThesisUtil
com.thesis.aop.util
xssFixOptions: String[]
sqlInjectionFixOptions: String[]
ThesisUtil()
doXSSFix(String,String):String
doXSSFix(byte[],String):byte[]
doSQLInjectionFix(Logger,String,String):String
getSolution(String,String):String
getSolution(Logger,String,String):String
```

Inside getSolution Method....

```
//If solution desired is SQL-ENCODE-MYSQL
else if(solution.equals(sqlInjectionFixOptions[0])){
    result = SQLInjectionValidation.escapeMySQL(s, logger);
}
//If solution desired is SQL-ENCODE-ORACLE
else if(solution.equals(sqlInjectionFixOptions[1])){
    result = SQLInjectionValidation.escapeOracle(s, logger);
}
```

Figure 32 - ThesisUtil Diagram and getSolution method logic

2. The SQLInjectionValidator class will first test the incoming query for comments. If the query has any comments, which could be used maliciously to bypass form validation, the comment is removed. This piece is done by using a pattern generated by a regular expression that uses a matcher to determine if the query has either single or multi line comments. The validator class then sends the intercepted query to the SQLParser class that will determine what kind of query was provided and pass it on to the JSQLParser algorithm. The types of queries that are supported are SELECT, INSERT, DELETE, and UPDATE. The JSQLParser library is an open source library that uses the visitor pattern to translate SQL statements using the Java Compiler Compiler (JavaCC) grammar for SQL [50]. The CCJSqlParserManager class is used to parse a string that represents an SQL query and it will return an object for the type of query it determines.

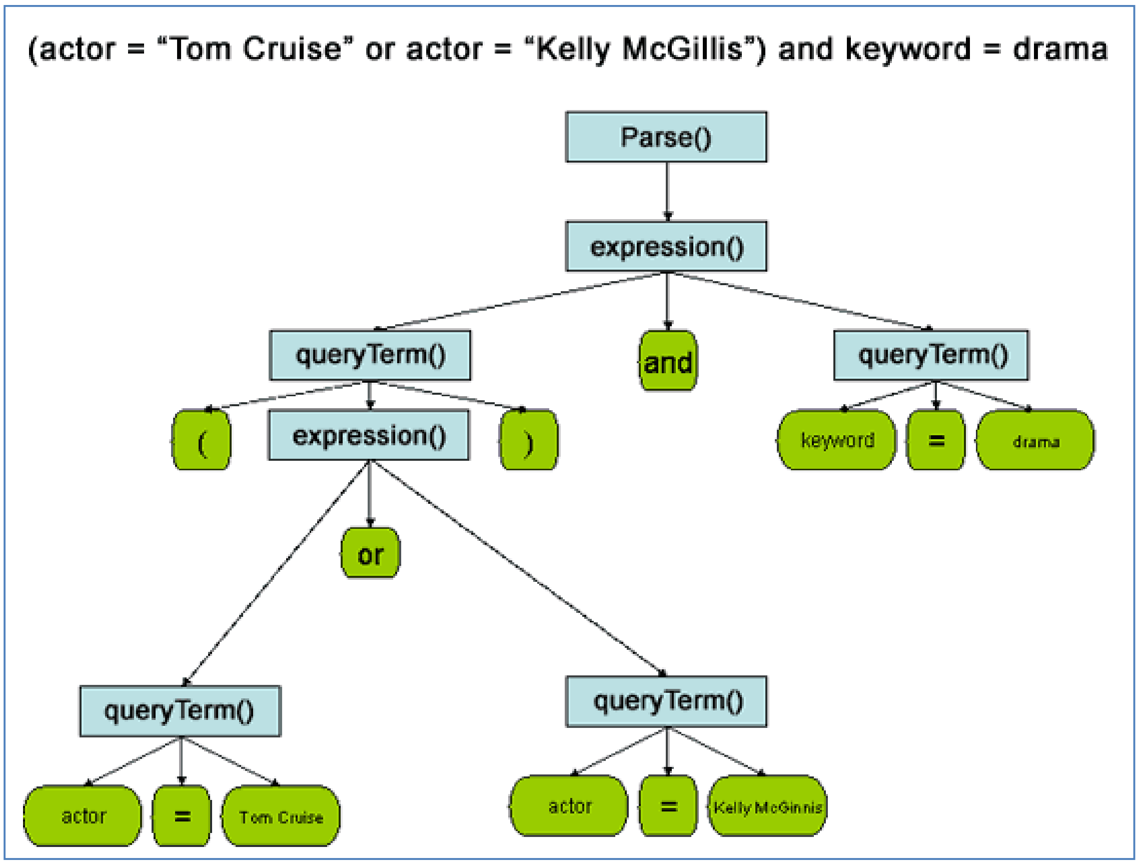


Figure 33 - How JavaCC Parses a query [50]

3. The JSQLParser will take the object determined by the CCJSqlParserManager and use the WhereItemsFinder class to generate a list of SimpleExpression objects that will be used by the validator class to encode. The WhereItemsFinder uses the visitor pattern and accepts different types of query objects whether they are instances of Select, Update, Insert, or Delete. The list of expressions that are supported by this class consists of:
    - GreaterThan – i.e. "10 > 5"
    - GreaterThanEquals – i.e. "10 >= 5"
    - MinorThan – i.e. "5 < 10"
    - MinorThanEquals – i.e. "5 <= 10"
    - IsNullExpression – i.e. "name = NULL" or "name != NULL"
    - NotEqualsTo – i.e. "name <> 'Homer'"
    - EqualsTo – i.e. "name = 'Homer'"

- Function – i.e. “RAND() > 1”

The visitor pattern implementation also has support for data types such as String, long, double, Date, TimeStamp, and others. The class will visit each of the expressions and create SimpleExpression objects that will have the expression’s left item (typically a column name), operator, right item (typically a parameter), and valueType of the parameter.

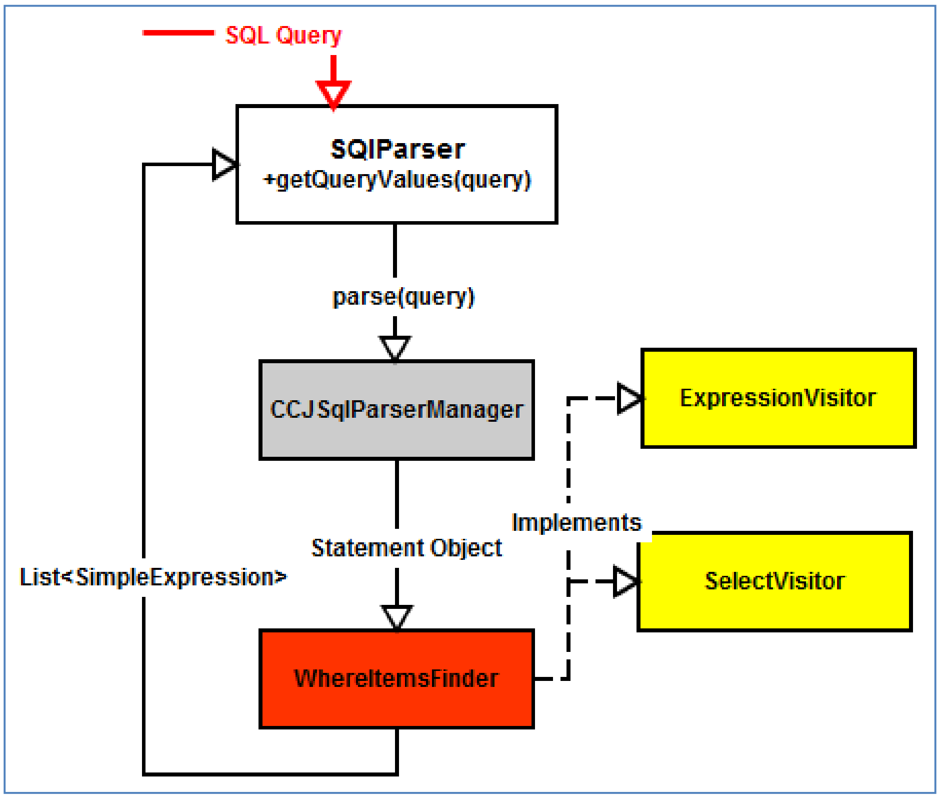


Figure 34 - SQL Parser flow diagram

4. The list of SimpleExpression objects will be returned to the SQL Injection Validator after each expression has been visited and analyzed. In the event that an expression contains an operator or function that is not supported, it will return the default value.
5. Once the potentially malicious query has been parsed and each expression object is available, the next task is to encode them individually. The encoding process will consist of iterating through each expression and using either the MySQL or Oracle codecs to do the encoding using either the escapeMySQL or

escapeOracle methods, respectively. The encoding will only be done if the value type of the expression is a string because there is no reason to encode integers because they cannot contain malicious scripts or data. As the list of SimpleExpression's is iterated, the string replace method will be used to insert the encoded value in place of the old one. The method executed to do the escaping will then return an encoded version of the query.

```
public static String escapeOracle(String s, Logger logger)
            throws JSQLParserException {
      OracleCodec oracle = new OracleCodec();
      String encodedResult = s;

      try {
      List<SimpleExpression> items = SQLParser.getQueryValues(s);
      for (Iterator iterator = items.iterator(); iterator.hasNext();) {
            SimpleExpression simpleExpression =
            (SimpleExpression) iterator.next();
            boolean tautology = testTautology(simpleExpression, logger);
            if (tautology) {
                  //Tautology Code
            } else if (simpleExpression.valueType == CONSTANT.VALUE_STRING) {
                  encodedResult = encodedResult.replace(
                              "\"" + simpleExpression.value + "\"",
                              "\"" + ESAPI.encoder().encodeForSQL(oracle,
                              simpleExpression.value) + "\"");
                  encodedResult = encodedResult.replace(
                              "'" + simpleExpression.value + "'",
                              "'" + ESAPI.encoder().encodeForSQL(oracle,
                              simpleExpression.value) + "'");
            }
      }
      } catch (JSQLParserException e) {
            logger.info("ERROR - INVALID QUERY ", e);
            encodedResult = "INVALID QUERY - CHECK LOGS";
      }
      return encodedResult;
}
```

Figure 35 - Encoding Algorithm for Oracle SQL Encoding

6. Once the possibly malicious query is encoded, it will be returned to the SQL Mitigation Aspect and executed using the proceed method in the advice. The parameter of the proceed(encodedString, …) method is now the newly encoded and presumably safe query. The aspect will log the successful execution of the SQL validation class with the execution time and type of encoding completed.

This section has described the implementation for mitigating SQL Injection vulnerabilities by using the ESAPI encoder library to validate all the expressions in a query. This algorithm is designed to only work with SELECT, INSERT, DELETE, and UPDATE queries. It also analyzes the "where" clauses of each of these and encodes any expressions within them. In the event that the query parsing fails, the algorithm will execute the query as is in order to preserve execution flow of the legacy application.

3.3.2 SQL Injection and Tautology Detection

In the process of mitigating SQL Injection vulnerabilities, it became necessary to create a tautology detector that would prevent one of the more common attacks from being executed. The tautology detection is executed for each SimpleExpression in the database encoding algorithms above. In the event of a tautology being detected in the query, the expression will be replaced by a value such as "1=2" to that it will no longer be true. The algorithm for detecting the tautology will be implemented in the SQL Injection Validator's testTautology method that will take a SimpleExpression parameter to test if the object contains a tautology.

The algorithm will implement the ScriptEngineManager library and set the script engine to use the JavaScript language. The reason that the JavaScript engine was chosen is because the language does not require data types to be defined when evaluating expressions since it is loosely typed. The algorithm will use the expressions column name, operator, and value attributes to determine if the expression passed in is a tautology. The expression will be evaluated using the ScriptEngine's eval method. If the result of the evaluation returns a true value, then the SQL Injection Validator will replace the tautology and insert a log entry that a tautology was detected and that it has been replaced. The testTautologyMethod is implemented as follows:

```
public static boolean testTautology(SimpleExpression exp,
            Logger logger) {
      ScriptEngineManager manager = new ScriptEngineManager();
      ScriptEngine engine = manager.getEngineByName("js");
      boolean resultBool = false;

      try {

      Object result;
      if (exp.getOp().equals("=")) {
            result = engine.eval("'" + exp.getColumnName() + "'"
                        + "==" + "'" +
                        exp.getValue() + "'");
      } else if (exp.getOp().equals("<>")) {
            result = engine.eval(exp.getColumnName() + "!="
                        + exp.getValue());
      } else {
            result = engine.eval(exp.getColumnName() + exp.getOp()
                        + exp.getValue());
      }

      result = (Boolean) result;

      Boolean boolResult = new Boolean(result.toString());

      System.out.println(result);
      resultBool = boolResult.booleanValue();

      } catch (ScriptException e) {
            logger.info("ERROR in Tautology Detection", e);
      } finally {
            return resultBool;
      }

}
```

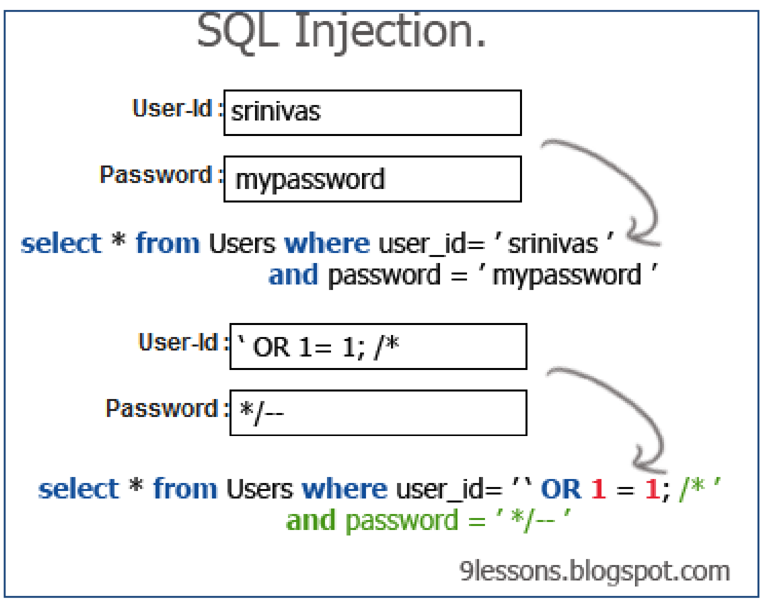
Figure 36 - testTautology Method Implementation in SQL Injection Validator

The SQL Injection Validator will then put the expressions' column name, operation, and value values to generate a regular expression. This regular expression will then be applied to the SQL query and replaced by "1=2" that is not a tautology and would eventually prevent the query from being true every time.

```
if (tautology) {
     logger.info("Obvious Tautology Detected with Left Side: "
          + simpleExpression.getColumnName()
          + " and Right Side: " + simpleExpression.value);

     logger.info("Replacing Obvious Tautology With '1=2' in
          order " + "to prevent execution of attack");
     String regex = "(.{0,1}" + simpleExpression.columnName
          + ".{0,1}\\s{0,1000}" + simpleExpression.op +
          "\\s{0,1000}.{0,1}" +
          simpleExpression.value + ".{0,1})";
     encodedResult = encodedResult.replaceAll(regex, " 1=2");
}
```

Figure 37 - Tautology replacement logic

The tautology detector was created to capture simple tautologies and not allow them to be executed as part of the SQL query. This was done because many times SQL Injection attacks such as the one below [51] will contain a tautology in order to bypass form validation. Since the tautology detector uses the JavaScript engine to evaluate expressions, there is no support for evaluation SQL functions such as RAND(). The tautology detector could be extended to support these scenarios but it was outside the scope of this paper.



```
select * from Users where user_id= ' srinivas '
          and password = ' mypassword '

select * from Users where user_id= '' OR 1 = 1; /* '
          and password = ' */-- '
```

Figure 38 - Example of login bypass SQL Injection Attack

3.3.3 XSS Mitigation Algorithms

The difficulty in preventing Cross Site scripting comes from the fact that there‘s such a large number of attack vectors available. The attacker could potentially steal the session of a victim, manipulate files on the victims’ computer, record all keystrokes the victim makes in a web application, and probe a company’s intranet where the victim is located [52]. Appropriate validation and escaping can address most reflected and stored XSS vulnerabilities. The algorithms described in this section will make use of the ESAPI encoder and validator libraries to do the escaping/encoding of malicious data.

Since most untrusted data comes from places such as URL parameters, headers, cookies, headers, and form fields that are placed on the HTTP Request object, it needs to be validated before it is stored anywhere. Stored XSS attacks can be executed by malicious content that is stored in the user database, web services, or other locations such as data files. Functions such as HTTPRequest.getParameter and println were identified by the Fortify SCA as potentially reading or writing data that is not validated. This papers’ XSS Validation aspect will create pointcuts for these functions and apply the fix selected by the user for each join point identified by the pointcuts as illustrated by the diagram below.

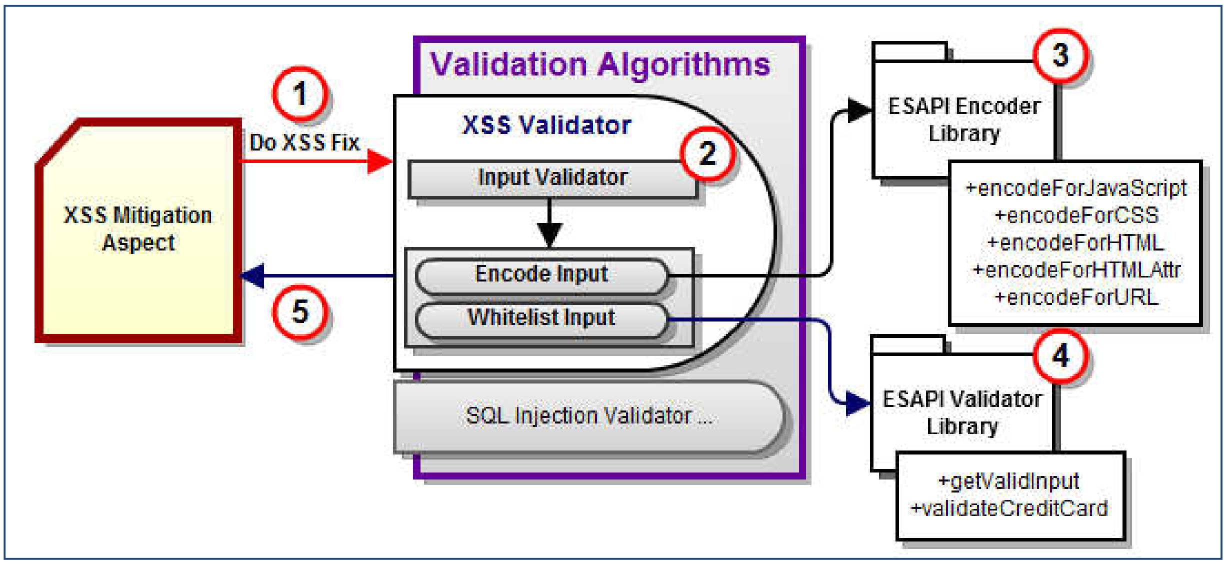


Figure 39 - XSS Validator Execution Flow

1. In the first step the XSS mitigation aspect's advice created by the AspectGenerator will call the doXSSFix method on the ThesisUtil class which will determine which type of fix to apply to the join point.
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 a whitelist. The user has the option to choose from one of these fixes for each join point [53] [54]:
    - JavaScript Encoding – Encode data for insertion inside a data value or function argument in JavaScript.
    - CSS Encoding – Encode data for use in CSS content.
    - HTML Attribute Encoding – Encode data for use in HTML attributes.
    - URL Encoding – Encode for use in a URL.
    - Credit Card Validation – Returns a canonicalized and validated credit card number as a String.
    - Alpha Numeric Whitelist – Apply whitelist to input to only accept alphanumeric strings.
    - Alpha Whitelist – Apply whitelist to input to only accept alpha strings.
    - Email Whitelist – Apply whitelist to input to only accept an email address formatted string.
    - Zip Code Whitelist – Apply whitelist to input to only accept a zip code formatted string.
    - IP Address Whitelist – Apply whitelist to input to only accept an IP address formatted string.
    - SSN Whitelist – Apply whitelist to input to only accept an SSN formatted string.
    - Custom Whitelist – Apply a custom whitelist via regular expression. The user will be able to provide a regular expression that will be enforced by the validator library.
    - Do Nothing – Execute join point regularly without doing any extra validation.
3. In this step the ESAPI encoder library is used to do the necessary encoding on the string that needs to be validated.

4. In this step the ESAPI validator library is used to whitelist a string based on the provided regular expression. If the string does not match the regular expression provided, the XSS Validator will throw an exception and then use a pattern matcher to remove the characters that do not match the provided regex.
5. The last step is for the XSS Validator to return the validated string back to the aspects' advice. The advice will then call the proceed method and pass it the encoded string.

The most difficult aspect of implementing the XSS Validator class is to catch all possible exceptions that the ESAPI encoder and validator classes can throw. In the event that an exception occurs, the program must handle it gracefully because the aspect is not supposed to break application execution flow. This means that if the aspects' advice causes an exception, the request is supposed to continue execution as normal.

3.3.4 SQL Injection Prevention JUnit Testing

The following section will discuss the JUnit testing code implementation for the program's SQLI and XSS Validator code. This is done so that the algorithms created can be tested thoroughly with the different possible combinations of malicious data that could be passed to each of the join points that are identified by the aspects' point cuts. First the section will discuss the implementation for the SQL Injection Validator and then the XSS Validators' test classes. The JUnit testing framework was selected for this task because it is integrated into the Eclipse IDE which made it simple to generate annotated stubs for each method in the test classes needed to be created.

To prove that the ESAPI database encoding and tautology detector are working as intended, it is necessary to do unit testing for the SQL Validator class and any classes that utilize it such as the ThesisUtil. To do this, JUnit4 tests are created for the Oracle and MySQL encoding functions in the SQL Validator. These tests are then executed with various types of attacks as well as normal inputs and results are provided.

The first task in implementing unit testing was to create a testing class for the SQL Injection Validator. The class created is called SQLInjectionValidationTestCase and it contains a constructor and a testing method with the “@test” annotation for each of the validators’ methods. The test class will also make use of the Log4j logging library to print out the results and execution times of each method to a log file named “unittests.log”. Each of the classes’ methods will use a String attribute defined in the test class which will represent a database query and execute the corresponding method in the SQL Validator class. If the result of the encoding contains an error string or the Validator execution fails, then the error will be logged. At the end of the test methods’ execution the StopWatch classes’ elapsed time will be logged along with the type of test that was just completed. The method code implementation for the testEscapeMySQL is shown below.

```
@Test
public void testEscapeMySQL() throws Exception {
	StopWatch t = new StopWatch();
	logger.info("Running MySQL Test");
	t.start();
	try {
		String result =
		SQLInjectionValidation.escapeMySQL(query, logger);

		if (result.contains("VALIDATION_FAILURE")) {
			logger.info(result);
		} else
			logger.info("SUCCESS: " + result);
	} catch (Exception e) {
			e.printStackTrace();
			logger.info("Exception in MySQL Test - FAILURE");
			fail("MySQL Test Failed - Exception Thrown");
	} finally {
			t.stop();
	}

	logger.info("Test took: " + t.getElapsedTime() +
			" ms");
	logger.info("MySQL Test Finished");
}
```

Figure 40 - JUnit Test Method for encodeMySQL

The XSS Validator code implementation is similar to the SQLI one in the sense that it uses the same JUnit library, logging functionality, and the same log file. The class created to do the unit testing for the XSS Validator is called XSSValidationTest and it implements test methods with annotations for the Validators' seven methods. Each of the test methods will try to execute the corresponding escape/validation method and will log the result. Each of the test methods will handle support for catching a general exception or a ValidationException that could be thrown by the ESAPI Encoder or Validator libraries. In the event of an exception, the test will fail, log the error, and print out the exception's stack trace as well. An example for the XSS Validators escapeCustomString method can be found below.

```
@Test
public void testEscapeCustomString() {
     String returnString = "";
     try {
          returnString =
          XSSValidation.escapeCustomString("abc123def456ghij-",
          "[a-z]+", 300);
     } catch (IntrusionException e) {
          // TODO Auto-generated catch block
          logger.info("XSS Test FAIL - EscapeCustomString " +
          returnString);
          logger.info("XSS Test FAIL - ", e);
          fail("IntrusionException");
     }
     catch (Exception e){
          logger.info("XSS Test FAIL - EscapeCustomString " +
          returnString);
          logger.info("XSS Test FAIL - ", e);
          fail("Exception");
     }
     logger.info("XSS Test Successful - EscapeCustomString " +
     returnString);
}
```

Figure 41 - JUnit test for XSS Validator escapeCustomString method

To take these tests a bit further, a JUnit test class was created for the projects' ThesisUtil class as well. This was done because the SQLI and XSS prevention aspects call this classes' doXSSFix and doSQLInjectionFix methods directly which in turn execute the two Validator algorithms. The test class for the ThesisUtil simply has methods for testing the two different types of fixes using the same logic as the above implementations and the same logging file.

3.4 Aspect Generation and Deployment

This section will explain the process that needs to be executed in order to create the XSS and SQL Injection mitigation aspects. The Eclipse project will be compiled and exported as an executable jar file. The jar file will be called XSS-SQLI-Mitigator.jar and can be executed via the java –jar command. When executing the jar file, the user will also need to specify the Fortify SCA report file in xml format. The remainder of this section will discuss how to use the aspect generator JAR file and how to deploy the resulting aspects with an existing or new web application.

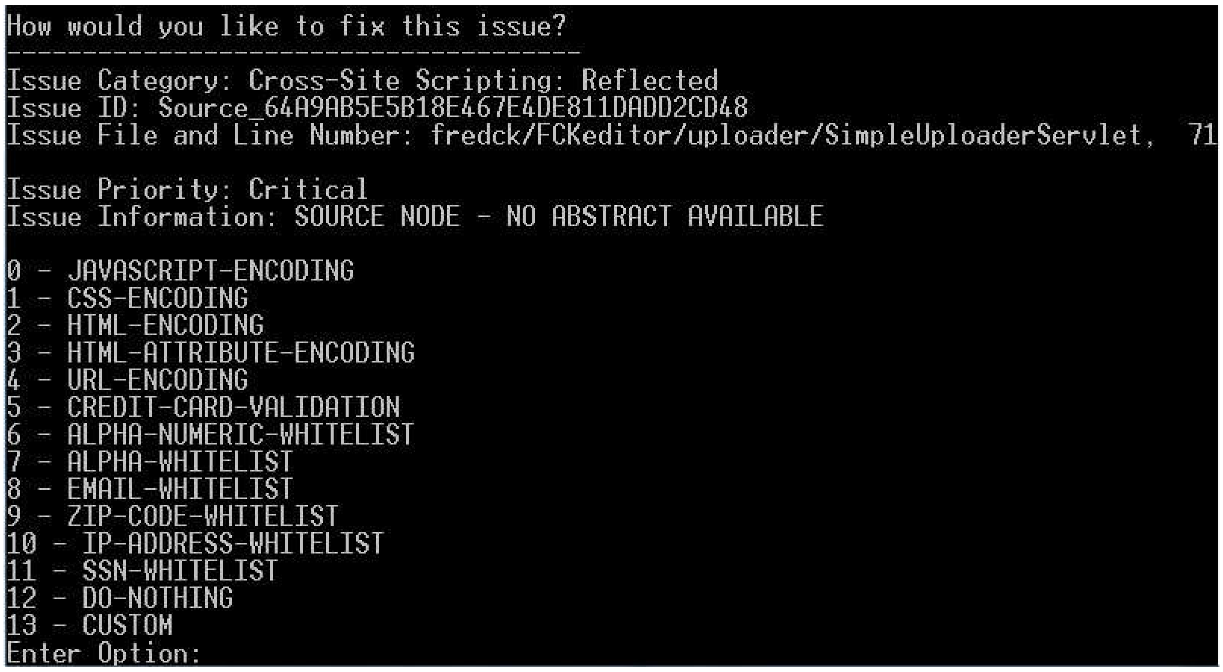
3.4.1 Using the Aspect Generator

The first step to executing the jar file is to open a command prompt or terminal window and go to the directory where the XSS-SQLI-Mitigator.jar file is located. Next, the jar file can be executed using the following format:

```
java -jar XSS-SQLI-Mitigator.jar <sca_report_xml_file>
```

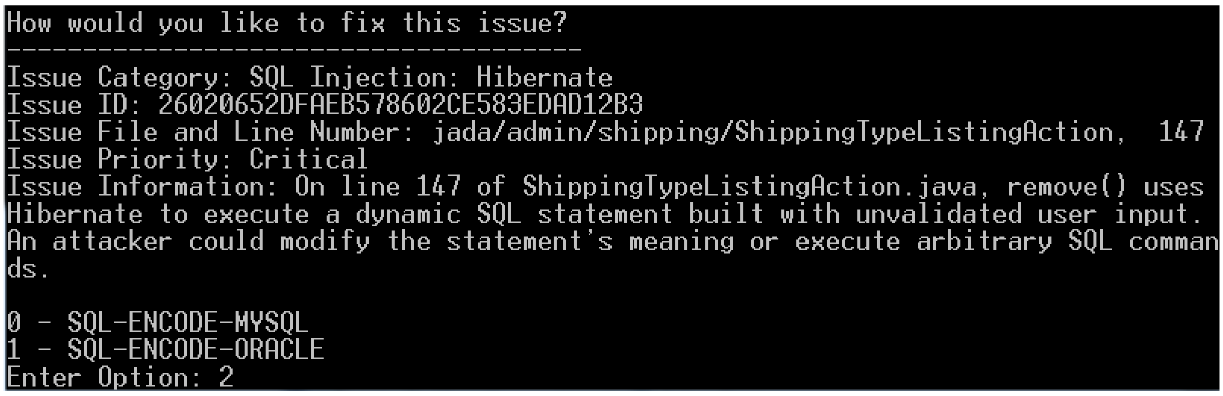
The <sca_report_xml_file> parameter will be the xml file generated by the Fortify Audit Workbench program described earlier in the chapter. Once the file is specified and command executed, the user will be asked to provide the desired fix for each of the vulnerabilities in the Fortify SCA report. The user will also be given specific information provided by the source code analyzer to guide them in making their decision for what fix to implement. This information includes the issue category, issue ID, file name, line number within that file, priority, and the abstract information. The reason that this approach was takes is to ensure the least amount of false positives because the user running this program should be someone with great understanding of the application.

In the case of XSS, the user will be provided the list of 14 options and will put in the corresponding digit. This requires substantial knowledge of the application and secure coding, which may be unavoidable but is still a disadvantage. If the user chooses the "CUSTOM" option they will be required to put in a regular expression that will be applied to whitelist the possibly malicious user input for that vulnerability. The SQL Injection vulnerabilities will have a list of only 2 possible fixes since the ESAPI library only supports database encoding for MySQL and Oracle implementations. There is also an implemented "auto pilot" feature for testing purposes. This can be enabled by putting in the number 99 as input at any time. This will then choose a fix at random from the available options.



```
How would you like to fix this issue?
--------------------------------------
Issue Category: Cross-Site Scripting: Reflected
Issue ID: Source_64A9AB5E5B18E467E4DE811DADD2CD48
Issue File and Line Number: fredck/FCKeditor/uploader/SimpleUploaderServlet,  71

Issue Priority: Critical
Issue Information: SOURCE NODE - NO ABSTRACT AVAILABLE

0 - JAVASCRIPT-ENCODING
1 - CSS-ENCODING
2 - HTML-ENCODING
3 - HTML-ATTRIBUTE-ENCODING
4 - URL-ENCODING
5 - CREDIT-CARD-VALIDATION
6 - ALPHA-NUMERIC-WHITELIST
7 - ALPHA-WHITELIST
8 - EMAIL-WHITELIST
9 - ZIP-CODE-WHITELIST
10 - IP-ADDRESS-WHITELIST
11 - SSN-WHITELIST
12 - DO-NOTHING
13 - CUSTOM
Enter Option:
```

Figure 42 - Asking user to provide XSS Fix type



```
How would you like to fix this issue?
--------------------------------------
Issue Category: SQL Injection: Hibernate
Issue ID: 26020652DFAEB578602CE583EDAD12B3
Issue File and Line Number: jada/admin/shipping/ShippingTypeListingAction,  147
Issue Priority: Critical
Issue Information: On line 147 of ShippingTypeListingAction.java, remove() uses
Hibernate to execute a dynamic SQL statement built with unvalidated user input.
An attacker could modify the statement's meaning or execute arbitrary SQL comman
ds.

0 - SQL-ENCODE-MYSQL
1 - SQL-ENCODE-ORACLE
Enter Option: 2
```

Figure 43 - Asking user to provide SQLI fix type

Once the user provides all the fixes, the program will generate the two aspects that will mitigate SQL Injection and XSS vulnerabilities for the application that the SCA report was provided for. The aspect files, “XSSAspect.aj” and “SQLInjectionAspect.aj” will be stored in the same directory as the executed jar file. The next step will be for the user to take these files and incorporate them into their application.

3.4.2 Deploying Generated Aspects

After the aspects are generated, they will need to be applied to the build path of the project that they need to be implemented onto. The user will then need to use the AspectJ compiler (AJC) to compile the aspects with the projects’ source code. The AJC will compile the .java and .aj files resulting in compiled bytecode that will have the “weaved” aspect code [55]. At runtime, the project can use its existing runtime but the application will now have its class files weaved with the aspect code.

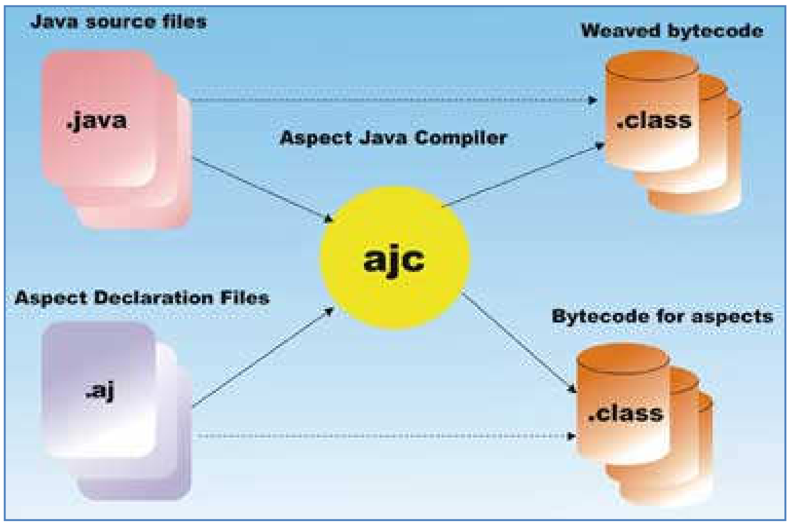


Figure 44 - AspectJ Compiler Execution [55]

When the user downloads the AspectJ compiler and unzips it to a directory on their workstation, there are a set of steps that make the compilation process of their aspects and application much simpler.

1. Keep their existing java project setup as is.
2. Create a separate project directory just for their aspects.

3. Use Ant to run the AJC on their existing source code and provide the output directory separate from the code base.
4. Run the application with the newly generated class files by the AJC.

The AJC can be executed either by the command line or using the AspectJ ANT tasks. Some of the variables that will need to be set are the ANT_HOME and JAVA_HOME environment variables. The AspectJ development toolkit (AJDT) also provides thorough documentation on building an application with the AJC. The Eclipse IDE also has a plug-in for the AJDT that allows the user to convert a project into an “AspectJ Project” which allows them to build the project directly from Eclipse.

3.5 Implementation Summary

This chapter has described the implementation of the SQL Injection and XSS mitigation aspects from the case study research to the aspect compilation and execution. For each case study chosen from various online open source repositories a Fortify source code scan was done to identify the vulnerabilities for the applications' source code. Once the vulnerabilities were identified they were exported into a readable XML format that could be utilized by the aspect creator program. This program used various templates and the users input to create aspects that would mitigate the vulnerabilities detected by the source code analysis. Several libraries and algorithms were implemented or created to be used by these aspects to mitigate vulnerabilities. The ESAPI library from OWASP was leveraged to apply encoding and validation for potentially malicious data and a tautology detecting algorithm was applied by leveraging the JSQLParser library. Unit tests were created for each of these algorithms using the JUnit framework to ensure correct execution and instructions were provided for executing the applications resulting executable jar file. Finally the chapter provided information for applying the programs generated aspects to their existing web application using the AspectJ compiler.

Chapter 4 Evaluation

This chapter will describe the evaluation of the work to create a program that generates aspects for mitigating SQL Injection and XSS in Java web applications. The overall desired results are to successfully prevent attacks on vulnerabilities identified by the Fortify Source Code Analyzer while not having to refactor any legacy source code. The evaluation will also need to prove that the libraries and algorithms used to eliminate two of the largest threats to application security are not only functional but also fast. Another important factor is that the security code is located in a centralized location and any updates to the security implementation with respect to SQLI and XSS can now be made in the aspects generated by this tool. 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. The chapter will conclude with the reflection of all results and how successful the evaluation was.

4.1 Methods

The evaluation of the aspects will need to consist of three different but equally important methods. First it will be important to evaluate all of the algorithms and libraries that the aspects will invoke at each join point chosen by the user. Then the actual aspects themselves will be evaluated in terms of how they are incorporated into different applications. Lastly 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. The evaluation will be done by creating JUnit4 tests for each of the libraries' methods and then running a stress test to see how well the libraries perform. The test for the SQL Injection validator will consist of running the encoding functionalities for the Oracle and MySQL ESAPI libraries' encoders and then evaluating the execution time and successful prevention of possible attacks. The tests will execute twenty-one potentially malicious queries 1000 times continuously and the results will be recorded into a log file which will then be analyzed. These queries are stored in the SQLITests.data file under the tests package in the application. They consist of queries that contain tautologies, comments, and data that needs to be encoded since it could contain malicious scripts.

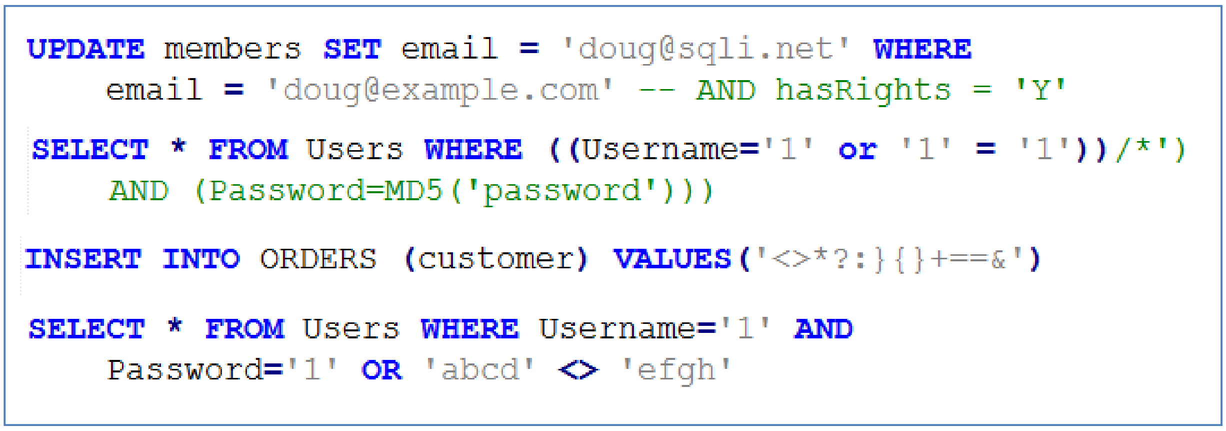
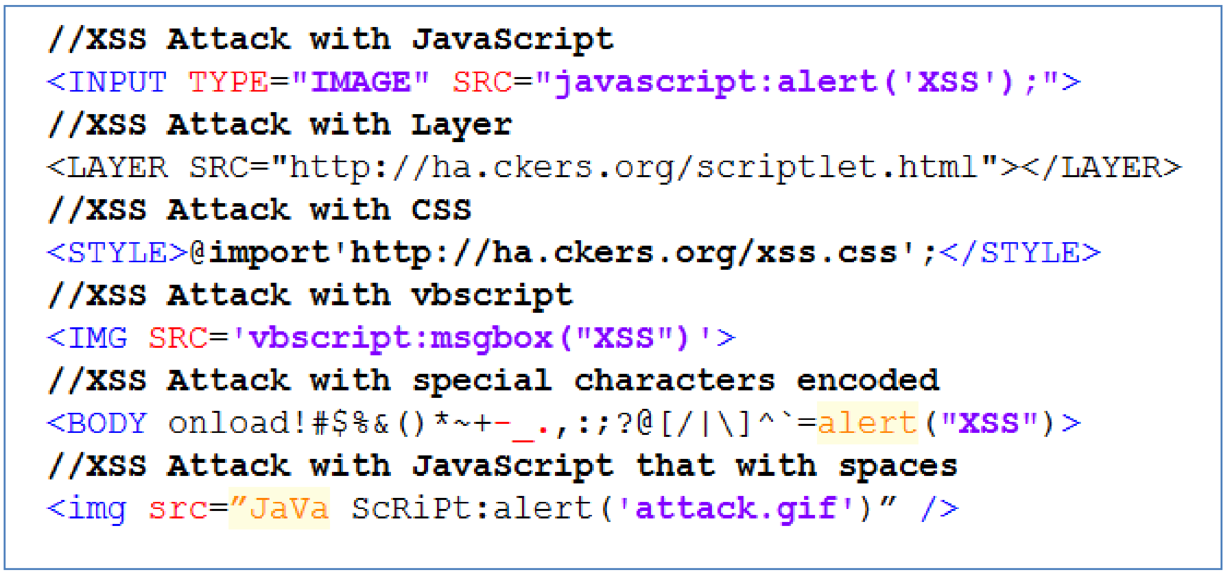

```
UPDATE members SET email = 'doug@sqli.net' WHERE
    email = 'doug@example.com' -- AND hasRights = 'Y'

SELECT * FROM Users WHERE ((Username='1' or '1' = '1'))/*')
    AND (Password=MD5('password')))

INSERT INTO ORDERS (customer) VALUES('<>*?:}{}+==&')

SELECT * FROM Users WHERE Username='1' AND
    Password='1' OR 'abcd' <> 'efgh'
```

Figure 45 - Potentially Malicious Queries in Data File

For each query executed, the log file will contain information such as the execution time, the type of fix that was implemented, and any changes that were made to the input. These changes could consist of character encoding, replacing tautologies detected, and removing comments from the query. In the event of a query being provided that is not of correct form and cannot be parsed, the original query is returned with a prepended error message to the join point.

The method for testing the XSS Validator is a bit more cut and dried than the logic applied to the SQLI Validator. This is because the user has the ability to apply a whitelist for each of the potentially malicious pieces of data captured at each join point. Since the whitelist approach is unforgiving, if an input does not match the whitelist regular expression, then it is rejected and a blank string is returned. As with the previous validator, the tests will be executed via JUnit4 that will run a series of potential inputs for each possible solution that the XSS Validator provides. These inputs consist of fifty-five different strings that will be passed to the validators' functions. The strings originate from the hackers.org XSS page that contains an extensive list of attack vectors for cross site scripting that has been replicated by the OWASP 2.0 guide [56]. To get an accurate execution time for each of the method executions, each method will be called 2500 times repeatedly and the results will be exported to a log file for analysis.



```
//XSS Attack with JavaScript
<INPUT TYPE="IMAGE" SRC="javascript:alert('XSS');">
//XSS Attack with Layer
<LAYER SRC="http://ha.ckers.org/scriptlet.html"></LAYER>
//XSS Attack with CSS
<STYLE>@import'http://ha.ckers.org/xss.css';</STYLE>
//XSS Attack with vbscript
<IMG SRC='vbscript:msgbox("XSS")'>
//XSS Attack with special characters encoded
<BODY onload!#$%&()*~+-_.,:;?@[/|\]^`=alert("XSS")>
//XSS Attack with JavaScript that with spaces
<img src=”JaVa ScRiPt:alert('attack.gif')” />
```

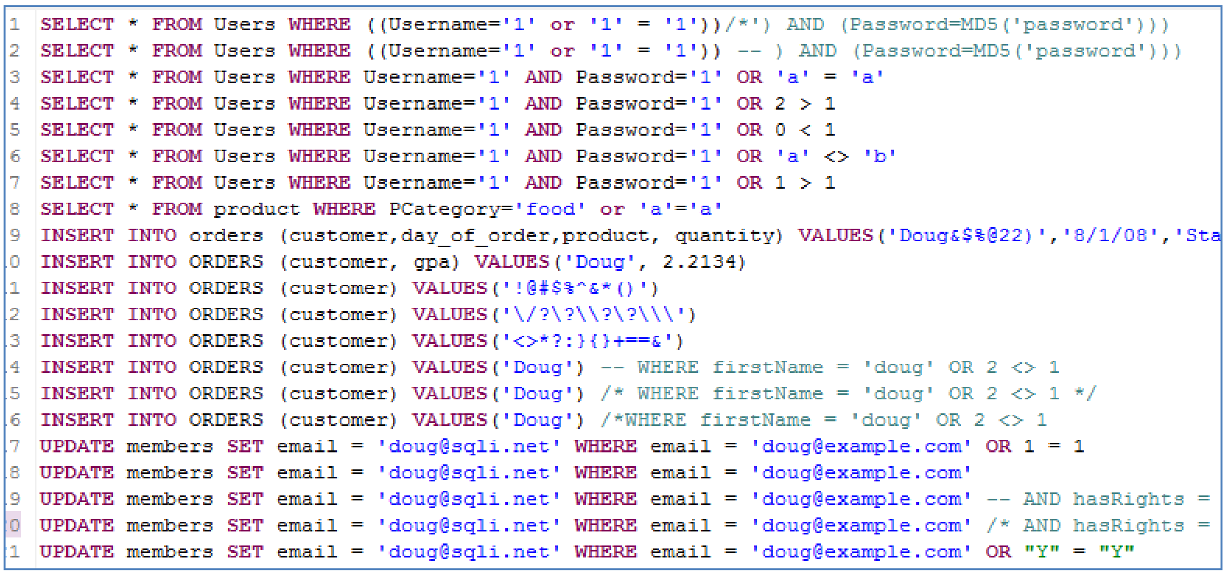
Figure 46 - Potential XSS attacks from XSS test data file.

The second method for evaluating the aspects generated will be to prove that the pointcuts created are actually affecting the locations of files identified by the Fortify SCA. For each of the selected case studies, the aspects will be weaved into the necessary source code files and the markers for the advice for each pointcut will then be compared to the vulnerabilities that the SCA detected. This step being successful will prove that the aspects will create join points that will be advised on correctly. Once all of the pointcuts are validated the last step will be to actually execute the application code and to make sure that the pointcuts are being executed and the correct results are being computed. Since some of the projects such as OfBiz, contain thousands of join points, the testing will only consist of some of them being tested due to time constraints. These time constraints come from the fact that the applications are so robust and that there are so many different possible paths to each join point, that it would be extremely time consuming to validate each one.

4.2 Results

This section will contain the results of the methods applied in the previous section. All of the stress tests done on the XSS and SQLI Validators will be executed on a Windows 7 machine with 8GB RAM and i5 Intel processor. The time values captured (in milliseconds) could vary depending on the hardware configuration of the machine that the tests are executed on.

For the first set of tests, a list of twenty-one SQL queries was executed fifty times for both the ESAPI MySQL and Oracle encoders. This list of queries did not exceed 300 characters and it contained different types of malicious content that could result in attacks ranging from injection of scripts to bypassing login forms.



```
SELECT * FROM Users WHERE ((Username='1' or '1' = '1'))/*') AND (Password=MD5('password')))
SELECT * FROM Users WHERE ((Username='1' or '1' = '1')) -- ) AND (Password=MD5('password')))
SELECT * FROM Users WHERE Username='1' AND Password='1' OR 'a' = 'a'
SELECT * FROM Users WHERE Username='1' AND Password='1' OR 2 > 1
SELECT * FROM Users WHERE Username='1' AND Password='1' OR 0 < 1
SELECT * FROM Users WHERE Username='1' AND Password='1' OR 'a' <> 'b'
SELECT * FROM Users WHERE Username='1' AND Password='1' OR 1 > 1
SELECT * FROM product WHERE PCategory='food' or 'a'='a'
INSERT INTO orders (customer,day_of_order,product, quantity) VALUES('Doug&$%@22)','8/1/08','Sta
INSERT INTO ORDERS (customer, gpa) VALUES('Doug', 2.2134)
INSERT INTO ORDERS (customer) VALUES('!@#$%^&*()')
INSERT INTO ORDERS (customer) VALUES('\/?\?\\?\?\\\')
INSERT INTO ORDERS (customer) VALUES('<>*?:}{}+==&')
INSERT INTO ORDERS (customer) VALUES('Doug') -- WHERE firstName = 'doug' OR 2 <> 1
INSERT INTO ORDERS (customer) VALUES('Doug') /* WHERE firstName = 'doug' OR 2 <> 1 */
INSERT INTO ORDERS (customer) VALUES('Doug') /*WHERE firstName = 'doug' OR 2 <> 1
UPDATE members SET email = 'doug@sqli.net' WHERE email = 'doug@example.com' OR 1 = 1
UPDATE members SET email = 'doug@sqli.net' WHERE email = 'doug@example.com'
UPDATE members SET email = 'doug@sqli.net' WHERE email = 'doug@example.com' -- AND hasRights =
UPDATE members SET email = 'doug@sqli.net' WHERE email = 'doug@example.com' /* AND hasRights =
UPDATE members SET email = 'doug@sqli.net' WHERE email = 'doug@example.com' OR "Y" = "Y"
```

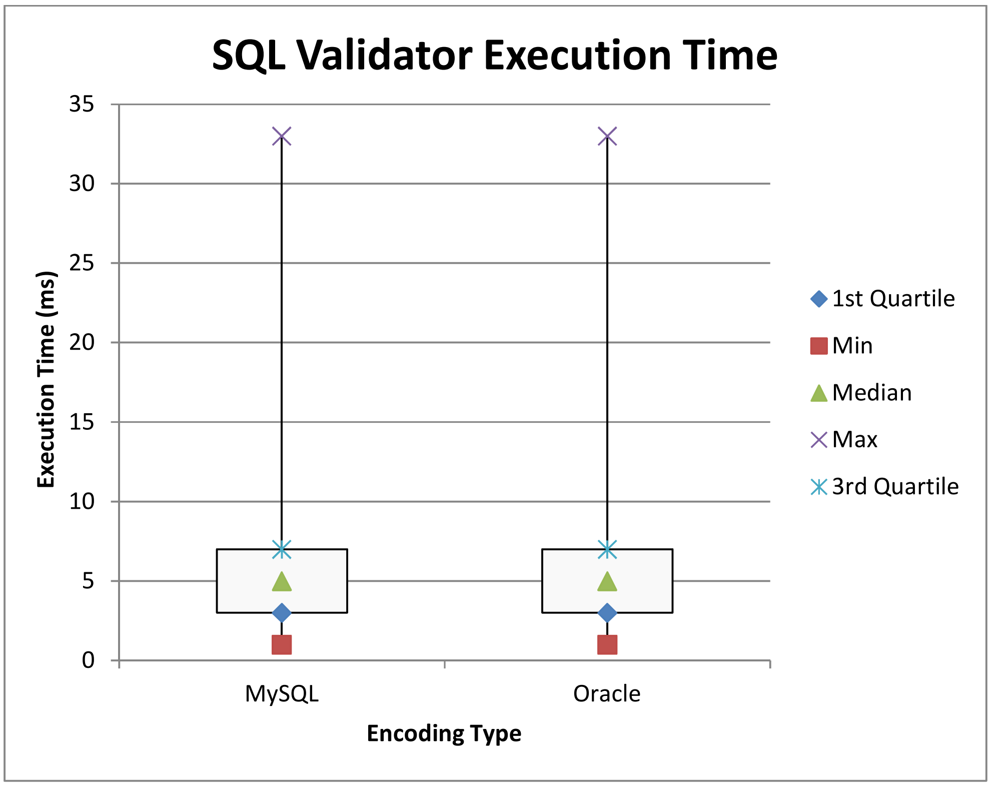
Figure 47 - List of queries tested for SQLI Validator

After the JUnit4 test was executed, the unitests.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 excellent time. 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. The result in the log file showed the following actions taken:

```
testLogger  :: Removing comment from query: "/*') AND
(Password=MD5('password')))" from "SELECT * FROM Users WHERE
((Username='1' or '1' = '1'))/*') AND (Password=MD5('password')))"
testLogger  :: Obvious Tautology Detected with Left Side: 1 and Right
Side: 1
testLogger  :: Replacing Obvious Tautology With 'x=y' in order to
prevent execution of attack
testLogger  :: ThesisUtil Success: result is - SELECT * FROM Users
WHERE ((Username='1' or  1=2))
testLogger  :: Test took: 123 ms
testLogger  :: testDoSQLInjectionFix Test Finished
```

As shown by this data, the malicious comment was removed from the query and the obvious tautology was replaced by something that would not evaluate to true such as '1=2'. The decision to replace the tautology was made because it is important to highlight to the user such events because then they can track them in their database logging efforts.

The second aspect of the SQLI Validator execution that was analyzed was the execution time for each of the query validations done. The data was collected from the log file and then the max, min, 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 result, the mode, was 5ms. The fact that the longest execution period was only 33ms was very encouraging because even though it was the highest value that only appeared a few times, it still was very short with respect to the execution time of the average web request. The computation time for the SQLI validation would most likely improve significantly if it was deployed on an enterprise level architecture with more robust hardware. Finally, the execution time for the MySQL and Oracle encoders were fairly identical where the max was the same at 33ms and min was 1ms.



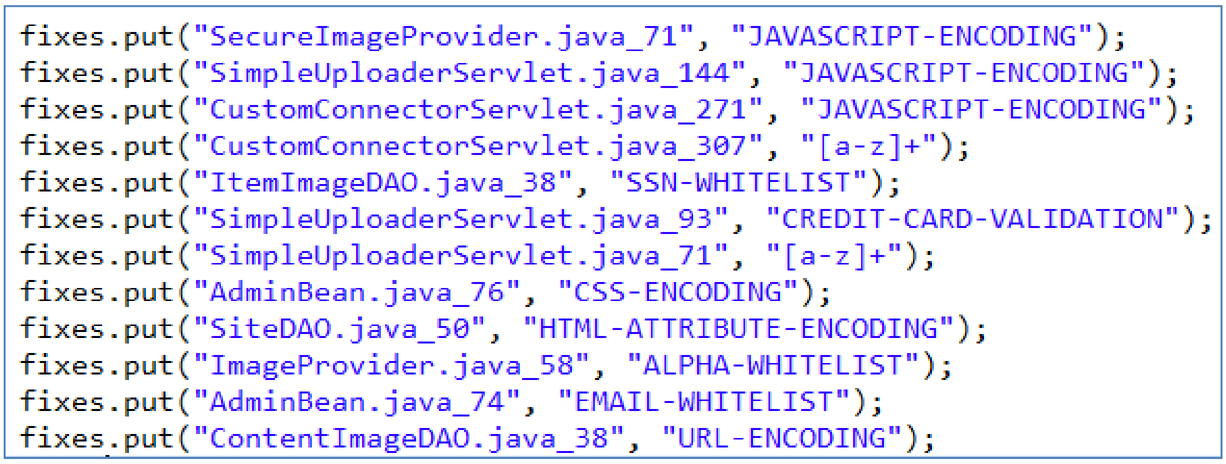
The evaluation results for testing the XSS Validator contained a considerably higher amount of information because of the many different fixes that are available to the user. However, the 13 different categories can be split into two separate categories that had similar results. The “encoding” category consists of fixes that use the ESAPI encoder library which simply encodes the users’ input characters. The second and generally more involved category is “whitelist”, where the users 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 mentioned above, the items in the category “encoding” 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 the 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 compares the strings’ value to see if it matches several different credit card patterns. There are several reasons for explaining the higher duration of the whitelist methods’ execution times. Each time the user requests to do whitelist validation of the input the program has to instantiate and compile a Pattern object for the corresponding regular expression. Then if the input string provided does not match the regular expression using the Matcher object, the sections of the string that do not match are replaced by an empty value. This approach ensures that the input provided is completely valid per the users’ request when they were asked to select a fix when the aspect generator program was executed.

The second portion of the evaluation is to integrate the SQLI and XSS aspects into the Alfresco, OfBiz, and JadaSite applications. The programs will be checked out from their respective SVN and CVS repositories and then imported as an eclipse project. For each project an “aspects” package will be created and configured into the build path of the application. The build path will also be configured to include the libraries and compiled class folder of the ThesisUtil, XSS Validator, SQLI Validator, and any other dependent classes. Once the build path is configured, each project will be built and the results will be analyzed. The analysis will consist of matching the list of vulnerabilities identified by the Fortify SCA with the markers for the advices identified by the AJDT compilation. The criteria for a successful evaluation will be the determination of each vulnerability identified having a corresponding join point for each of the solutions chosen by the user.

For the JadaSite application, the XSS and SQL Injection aspects had 12 and 109 vulnerabilities identified, respectively. For the SQL Injection aspect created, the pointcut that identified markers for the largest number of items was for the createQuery function with a total of 125. Although the SCA only identified a total of 110 SQL Injection vulnerabilities, the advice would only execute the SQLI Validator if the join point was on the exact line number the SCA specified. The executeQuery and createSQLQuery pointcuts both had only a single join point identified and the executeUpdate only had four. Since the JadaSite project did not use prepared statements, the pointcuts created for those had no advice markers and would therefore never be executed since it is not woven into any of the source code at compile time.



```
fixes.put("SecureImageProvider.java_71", "JAVASCRIPT-ENCODING");
fixes.put("SimpleUploaderServlet.java_144", "JAVASCRIPT-ENCODING");
fixes.put("CustomConnectorServlet.java_271", "JAVASCRIPT-ENCODING");
fixes.put("CustomConnectorServlet.java_307", "[a-z]+");
fixes.put("ItemImageDAO.java_38", "SSN-WHITELIST");
fixes.put("SimpleUploaderServlet.java_93", "CREDIT-CARD-VALIDATION");
fixes.put("SimpleUploaderServlet.java_71", "[a-z]+");
fixes.put("AdminBean.java_76", "CSS-ENCODING");
fixes.put("SiteDAO.java_50", "HTML-ATTRIBUTE-ENCODING");
fixes.put("ImageProvider.java_58", "ALPHA-WHITELIST");
fixes.put("AdminBean.java_74", "EMAIL-WHITELIST");
fixes.put("ContentImageDAO.java_38", "URL-ENCODING");
```

Figure 48 - Example of XSS Aspect fix map

The XSS Aspect for the JadaSite application only had 12 fixes identified in its fix map. Just like in the SQLI Aspect, the pointcuts identified more join points then there were fixes and the same logic applies in the sense that only join points where fixes are identified will be executed. The pointcut for the println function identified the most join points at 22 while the write byte array function only had one. Since the JadaSite project did not utilize the regular print, send error, or set attribute functions, these pointcuts did not have any markers and were therefore not woven into any parts of the projects' source code.

JadaSite	XSS Aspect	12	22
JadaSite	SQLI Aspect	109	131
OfBiz	XSS Aspect	130	380
OfBiz	SQLI Aspect	152	30
Alfresco	XSS Aspect	12	22
Alfresco	SQLI Aspect	13	13

The remainder of the results for each of the projects and corresponding aspects are displayed in the table above. To make sense between the discrepancies between the amounts of vulnerabilities identified by the SCA and the number of join points identified by the aspects, we have to analyze the difference between a sources and sink issue. Many times the Fortify SCA will identify multiple issues for the same source or sink of the vulnerability. An example is when a class invokes a method that contains the same vulnerability in multiple places. This is primarily evident in the SQLI Aspect of the OfBiz project where the number of vulnerabilities is 152 and only 30 join points were identified. In the event where the number of join points far exceeds the amount of vulnerabilities such as in the OfBiz XSS Aspect, this will not hinder performance because the advice has logic implemented that will only execute the XSS Validator if the join point location exists in the fix map.

The third and final part of the evaluation process was to test the aspects with all three case study applications running in a live environment. The difficulty of this validation step comes from the fact that each of the vulnerabilities identified by the SCA are scattered throughout the application and the path to each of these can be very difficult to reach since all the projects are enterprise level 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 and then analyze their execution process. When doing this step, the aspect generation program must either be extracted into a JAR or included into the build path of each of the applications so that the XSS and SQLI Validators as well as any required libraries such as the JSQL Parser can be referenced.

For the OfBiz project a total of 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 really consists of multiple projects with different contexts, the "repository" project was the one chosen for execution since it contained a fairly large portion of the vulnerabilities identified. This project contained 11 and 4 of the SQLI and XSS vulnerabilities, respectively. Both of the aspects executed as expected except one in the SQLI category where the query, doing table creation, was not supported by the JSQL Parser. However, even though the query was not supported, the program still executed as normal because the SQLI Validator returns the original value if query validation fails. The JadaSite project had no issue executing any of its 8 XSS and 14 SQLI join points that were tested.

4.3 Evaluation Conclusion

The evaluation of this project showed a successful implementation of the XSS and SQLI prevention aspects in three case study Java web applications. The first section of this chapter evaluated the execution time of the Validators that were created to mitigate SQL Injection and Cross-site scripting vulnerabilities. The results of this section proved that the application of mitigation algorithms on users input and query execution was very cheap in terms of length of execution and performance impact. The second section explained how the aspects generated were applied to each of the three case studies and how the join points identified by the AspectJ compiler correspond to the vulnerabilities identified by the Fortify SCA. This section determine that even in the event where the number of vulnerabilities was greater or less then the number of join points, each potentially malicious threat requested from the user was still handled. The third and final part of the evaluation was to execute the case studies and see the aspects' weaved code results. Even though it wasn't possible to test all vulnerabilities identified, the fact that the ones that were tested executed successfully is confidence enough that the rest would work as well when the corresponding join point was encountered.

Chapter 5 Conclusion and Future Works

5.1 Concluding Summary

Two of the largest threats to web application security 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 have a significant number of SQLI and XSS vulnerabilities that can be exploited by a malicious user. The SQLI and XSS mitigation aspect generator 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 validation to implement. The most significant feature of the approach identified is the use of the Aspect Oriented Programming paradigm that allows the code that mitigates vulnerabilities to be applied to existing applications without the need to modify potentially fragile source code.

The approach taken was to use the AOP paradigm to execute Validator classes that would be applied 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 the aspect generator, they will be given a choice of fixes to implement for each of the vulnerabilities detected by the SCA. The two resulting aspects, one for XSS and the other for SQLi, 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.

The XSS Aspect will handle the mitigation of Cross-site scripting vulnerabilities. The validation done will consist of either whitelisting or encoding of user input at the different join points identified by each pointcut. The user will have the capability to provide their own whitelist regular expression in the event that the default list of options is insufficient. The SQL Injection Aspect consists of advices that apply encoding to either Oracle or MySQL queries. The SQLI Validator also contains a tautology detector and comment parser that is used to thwart many malicious queries from being executed. The pointcuts for both aspects are defined in XML files that contain functions that need to be intercepted. These functions are created manually by analyzing the Fortify SCA report and identifying the code that is listed as having vulnerabilities. The aspects are then generated using their respective template files that contain placeholders that are replaced by values identified by the functions to be intercepted as well as their parameters. One of the restrictions of this approach is that the user executed aspect generator does not have the capability to expand the defined list of functions although it can be done by manually changing the XML configuration and regenerating the JAR file.

In order to evaluate the success of the aspects created, 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. Source code analysis was done on all three applications and the results were given to the aspect generator which provided security mitigation aspects. The evaluation consisted of doing extensive testing using the JUnit testing framework, integration of the 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 SQLI 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 thesis describes the implementation of a program that generates XSS and SQLI mitigation aspects that can be applied to existing web applications based on static source code analysis. The main advantage of this approach compared to others evaluated is the fact that the use of the AOP paradigm does not require any modification to the legacy code and provides a centralized location for the applications' security logic. The evaluation of the 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.

5.2 Future Works

There are a number of features and improvements that can be made to the aspect generator program that would make it a more effective security tool and provide a better overall user experience. The future works section of this paper will be broken down into four sections that will describe these updates, the reason behind each one, and how they would make the tool a better product. The first of these is to extend the program to support mitigation of more than just two types of vulnerabilities. The first step would be to implement aspect generation and input validation for the rest of the OWASP top 10 vulnerabilities as well as implement support for other types of source code analysis tools. The second step would be to make the pointcut and advice generation more extensible. This would allow the user running the aspect generator program to create customized pointcuts, advice templates, and have more control over what fixes to implement for the vulnerabilities identified. The third plan for future work is to create a Graphical User Interface (GUI) for the aspect generator program. This would enhance the users' experience using the aspect generator program by making it easier to use and more intuitive. The fourth improvement is the largest and most complex to implement. The desired result will be to extend the concept of this paper to other programming languages.

The first step would consist of identifying the other 8 top security vulnerabilities in web applications. For each additional type of potential attack to be handled by the aspect generator, the application would have to be extended to parse the corresponding input file from the source code analyzer. Template files would have to be created for each new aspect to be generated for both the pointcut and advice sections and functions to intercept would have to be identified and put into corresponding XML files. Once the template files are completed, algorithms would need to be written to implement the fixes that are identified by the source code analyzer. These algorithms should leverage any existing and proven methods for handling these various types of attack but this time they would be executed from the aspect. The second part of this first step would be to make the aspect generator compatible with more than just the Fortify source code analyzer. The goal would be to allow the user to provide a parameter to the program that would tell it what kind of SCA was used to generate the data file and the program should then use the corresponding data file parser. For the first set of updates, the program should be extended to support two more types of source code analysis tools.

The second step would be to allow the user to have greater control over what pointcuts are created in the aspect generation process. Currently the pointcuts are generated by the function definitions inside the xssfunctions and sqlInjectionFunctions XML files. The program will need to be modified to allow the user to create custom function entries for these files. The user will need to provide all the function data and parameters identified above and the entry will then be added to the XML file. Once the aspect generation is initialized, the users' entries will have pointcuts created for them for each aspect they provided additional function nodes for. The user would also need to have the ability to create all new aspect and pointcut templates that they could be used for the aspect generation instead of the standard ones. This would allow the user to use custom logging functionality, call other validation libraries, and possibly implement different advices such as before and after for each of their defined pointcuts. This update to the aspect generator would benefit the user because it would allow them to customize functionality of the aspect while still ensuring security. To make sure of this, certain parts of the aspects' template would have to remain the same.

The third enhancement to be implemented as part of the future works would be to create a GUI for the security aspect generator program. The GUI will most likely be a Java Swing application or an eclipse IDE plugin that would allow the user to interact with the program using the mouse and keyboard. The user would have controls such as buttons, dropdowns, lists, and text boxes for various actions they can perform. The user would be able to choose what fixes to implement from a drop down list and a text box to insert a custom regular expression in the case of a XSS whitelist fix. They would also have input fields for doing things such as extending the functions XML file and the pointcut and aspect template files defined above. The user interface would also have input validation which us currently non-existent in the command line implementation.

The effort to extend the program created to support multiple languages will be the most difficult because of the varying implementation of the aspect oriented programming paradigm. This is particularly different in interpreted programming languages such as PHP, Python, and Ruby. The first challenge to overcome would be to find a suitable AOP implementation for each new programming language that needs to be supported. Since the implementation for each language would be different, template files, function definitions, and libraries for mitigating vulnerabilities would need to be created for each one. The data used for the aspect generator for each of the newly supported language will need to come from a source code analyzer that has the ability to analyze projects of different programming languages. The initial phase of extending the security aspect generator for extra languages will be to provide support for PHP since there is such a large amount of web applications created using it. The challenge for supporting PHP will be to find an AOP implementation that does not require pre compilation of the files to weave in the necessary code.

With the above improvements and features added, the security aspect generator program will be much more robust and have a broader appeal to the development community. The program will also be available on a website such as GitHub where members of the open source programming community will be able to make additional feature requests as well as report any bugs they encounter.

Appendices

Appendix A SQL Injection Functions Definition XML

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<functions type="xml">
	<function name="createSQLQueryFunction">
		<methodName>createSQLQuery</methodName>
		<param type="String" name="param1" location="1" result="true" />
		<interceptParam>1</interceptParam>
		<interceptParamType>String</interceptParamType>
	</function>

	<function name="executeUpdateFunction">
		<methodName>executeUpdate</methodName>
		<param type="String" name="param1" location="1" result="true" />
		<interceptParam>1</interceptParam>
		<interceptParamType>String</interceptParamType>
	</function>

	<function name="executeQueryFunction">
		<methodName>executeQuery</methodName>
		<param type="String" name="param1" location="1" result="true" />
		<interceptParam>1</interceptParam>
		<interceptParamType>String</interceptParamType>
	</function>

	<function name="createQueryFunction">
		<methodName>createQuery</methodName>
		<param type="String" name="param1" location="1" result="true" />
		<interceptParam>1</interceptParam>
		<interceptParamType>String</interceptParamType>
	</function>

	<function name="prepareStatementFunction">
		<methodName>prepareStatement</methodName>
		<param type="String" name="param1" location="1" result="true"/>
		<interceptParam>1</interceptParam>
		<interceptParamType>String</interceptParamType>
	</function>

	<function name="prepareStatementFunctionMultiParam">
		<methodName>prepareStatement</methodName>
		<param type="String" name="param1" location="1" result="true"/>
		<param type="int" name="param2" location="2" result="false"/>
		<param type="int" name="param3" location="3" result="false"/>
		<interceptParam>1</interceptParam>
		<interceptParamType>String</interceptParamType>
	</function>

	<function name="prepareStatementFunctionTwoParamIntArray">
		<methodName>prepareStatement</methodName>
		<param type="String" name="param1" location="1" result="true"/>
		<param type="int" name="param2" location="2" result="false"/>
		<interceptParam>1</interceptParam>
		<interceptParamType>String</interceptParamType>
	</function>
```

```
      <function name="prepareStatementFunctionTwoParam">
            <methodName>prepareStatement</methodName>
            <param type="String" name="param1" location="1" result="true"/>
            <param type="int[]" name="param2" location="2" result="false"/>
            <interceptParam>1</interceptParam>
            <interceptParamType>String</interceptParamType>
      </function>

      <function name="generalExecuteFunction">
            <methodName>execute</methodName>
            <param type="String" name="param1" location="1" result="true"/>
            <interceptParam>1</interceptParam>
            <interceptParamType>String</interceptParamType>
      </function>

</functions>
```

Appendix B XSS Functions Definition XML

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<functions type="xml">
      <function name="writeFunction">
            <methodName>write</methodName>
            <interceptParam>1</interceptParam>
            <interceptParamType>String</interceptParamType>
            <param type="String" name="param1" location="1" result="true" />
      </function>

      <function name="writeByteArrayFunction">
            <methodName>write</methodName>
            <interceptParam>1</interceptParam>
            <interceptParamType>byte[]</interceptParamType>
            <param type="byte[]" name="param1" location="1" result="true" />
      </function>

      <function name="printFunction">
            <methodName>print</methodName>
            <interceptParam>1</interceptParam>
            <interceptParamType>String</interceptParamType>
            <param type="String" name="param1" location="1" result="true" />
      </function>

      <function name="printlnFunction">
            <methodName>println</methodName>
            <interceptParam>1</interceptParam>
            <interceptParamType>String</interceptParamType>
            <param type="String" name="param1" location="1" result="true" />
      </function>

      <function name="sendErrorFunction">
            <methodName>sendError</methodName>
            <interceptParam>2</interceptParam>
            <interceptParamType>String</interceptParamType>
            <param type="String" name="param1" location="1" result="true" />
      </function>
      <function name="getParameterFunction">
            <methodName>getParameter</methodName>
            <interceptParam>1</interceptParam>
            <interceptParamType>String</interceptParamType>
            <param type="String" name="param1" location="1" result="true" />
      </function>

      <function name="setAttributeFunction">
            <methodName>setAttribute</methodName>
            <interceptParam>2</interceptParam>
            <interceptParamType>String</interceptParamType>
            <param type="String" name="param1" location="1" result="true" />
      </function>

      <function name="writeByteArrayThreeParam">
            <methodName>write</methodName>
            <interceptParam>1</interceptParam>
            <interceptParamType>String</interceptParamType>
            <param type="byte[]" name="param1" location="1" result="true" />
            <param type="String" name="param2" location="1" result="false" />
            <param type="String" name="param3" location="1" result="false" />
      </function>
</functions>
```

Appendix C XSS Aspect Generated

```
package com.aspects;
import com.thesis.aop.util.ThesisUtil;
import com.thesis.aop.util.StopWatch;
import org.apache.log4j.Level;
import org.apache.log4j.Logger;
import org.apache.log4j.PropertyConfigurator;
import java.util.HashMap;
import org.owasp.esapi.errors.IntrusionException;
import org.owasp.esapi.errors.ValidationException;
import net.sf.jsqlparser.JSQLParserException;


public aspect XSSAspect{
   public HashMap<String, String> fixes = new HashMap<String, String>();
   public HashMap<String, String> customRegex =
      new HashMap<String, String>();
   Logger logger;
   StopWatch watch;

   public XSSAspect(){
      PropertyConfigurator.configure("log4j.properties");
      logger = Logger.getLogger("xssLogger");
      logger.info("XSS Aspect Created!");
      watch = new StopWatch();

      fixes.put("AVMWebappLoader.java_1451", "CSS-ENCODING");
      fixes.put("TestDeploymentTransportTransformer.java_73",
         "SSN-WHITELIST");
      fixes.put("BaseServlet.java_232", "CSS-ENCODING");
      fixes.put("BaseServlet.java_254", "ZIP-CODE-WHITELIST");
      fixes.put("AVMRemoteTransportService.java_411",
            "SSN-WHITELIST");
      fixes.put("GetMethod.java_431", "ZIP-CODE-WHITELIST");
      fixes.put("BaseServlet.java_283", "URL-ENCODING");
      fixes.put("WebDAV.java_447", "CSS-ENCODING");
      fixes.put("BaseServlet.java_284", "CSS-ENCODING");
      fixes.put("RepoRemoteTransportService.java_459",
            "JAVASCRIPT-ENCODING");
      fixes.put("SampleEncryptionTransformer.java_206",
            "CSS-ENCODING");
      fixes.put("GetMethod.java_405", "CSS-ENCODING");

   }


   pointcut xss_writeFunction(String param1) :
      call(* *.write(String)) &&
      (within(*..BaseServlet) ||
      within(*..AVMWebappLoader) ||
      within(*..AVMRemoteTransportService) ||
      within(*..RepoRemoteTransportService) ||
      within(*..WebDAV) ||
      within(*..GetMethod) ||
      within(*..SampleEncryptionTransformer) ||
      within(*..TestDeploymentTransportTransformer)) &&
      args(param1);

   Object around(String param1) : xss_writeFunction(param1){
      watch.start();
      if(fixes.containsKey(thisJoinPoint.getSourceLocation().getFileName()
            + "_" + thisJoinPoint.getSourceLocation().getLine())){

         try{
            String result = ThesisUtil.doXSSFix(param1,
               fixes.get(thisJoinPoint.getSourceLocation().getFileName()
               + "_" + thisJoinPoint.getSourceLocation().getLine()));
            return proceed(result);
         } catch (Exception e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
         }
      }
      else{
         return proceed(param1);
      }
      watch.stop();
      logger.info(thisJoinPoint.getSourceLocation().getFileName()
         + "_" + thisJoinPoint.getSourceLocation().getLine() + "," +
         watch.getElapsedTime());

      return proceed(param1);
   }

   pointcut xss_writeByteArrayFunction(byte[] param1) :
      call(* *.write(byte[])) &&
      (within(*..BaseServlet) ||
      within(*..AVMWebappLoader) ||
      within(*..AVMRemoteTransportService) ||
   within(*..RepoRemoteTransportService) ||
   within(*..WebDAV) ||
   within(*..GetMethod) ||
   within(*..SampleEncryptionTransformer) ||
   within(*..TestDeploymentTransportTransformer)) &&
   args(param1);

Object around(byte[] param1) : xss_writeByteArrayFunction(param1){
   watch.start();
   if(fixes.containsKey(thisJoinPoint.getSourceLocation().getFileName()
      + "_" + thisJoinPoint.getSourceLocation().getLine())){

      try{
         byte[] result = ThesisUtil.doXSSFix(param1, fixes.get(
            thisJoinPoint.getSourceLocation().getFileName() + "_" +
            thisJoinPoint.getSourceLocation().getLine()));
         return proceed(result);
      } catch (Exception e) {
         // TODO Auto-generated catch block
         e.printStackTrace();
      }
   }
   else{
      return proceed(param1);
   }
   watch.stop();
   logger.info(thisJoinPoint.getSourceLocation().getFileName() + "_" +
      thisJoinPoint.getSourceLocation().getLine() +
      "," + watch.getElapsedTime());

   return proceed(param1);
}

pointcut xss_printFunction(String param1) :
   call(* *.print(String)) &&
   (within(*..BaseServlet) ||
   within(*..AVMWebappLoader) ||
   within(*..AVMRemoteTransportService) ||
   within(*..RepoRemoteTransportService) ||
   within(*..WebDAV) ||
   within(*..GetMethod) ||
   within(*..SampleEncryptionTransformer) ||
   within(*..TestDeploymentTransportTransformer)) &&
   args(param1);

Object around(String param1) : xss_printFunction(param1){
   watch.start();
   if(fixes.containsKey(thisJoinPoint.getSourceLocation().getFileName()
      + "_" + thisJoinPoint.getSourceLocation().getLine())){

      try{
         String result = ThesisUtil.doXSSFix(param1,
            fixes.get(thisJoinPoint.getSourceLocation().getFileName()
            + "_" + thisJoinPoint.getSourceLocation().getLine()));
         return proceed(result);
      } catch (Exception e) {
         // TODO Auto-generated catch block
```

```
            e.printStackTrace();
         }
      }
      else{
         return proceed(param1);
      }
      watch.stop();
      logger.info(thisJoinPoint.getSourceLocation().getFileName()
      + "_" +
      thisJoinPoint.getSourceLocation().getLine() + "," +
      watch.getElapsedTime());

      return proceed(param1);
   }

   pointcut xss_printlnFunction(String param1) :
      call(* *.println(String)) &&
      (within(*..BaseServlet) ||
      within(*..AVMWebappLoader) ||
      within(*..AVMRemoteTransportService) ||
      within(*..RepoRemoteTransportService) ||
      within(*..WebDAV) ||
      within(*..GetMethod) ||
      within(*..SampleEncryptionTransformer) ||
      within(*..TestDeploymentTransportTransformer)) &&
      args(param1);

   Object around(String param1) : xss_printlnFunction(param1){
      watch.start();
      if(fixes.containsKey(thisJoinPoint.getSourceLocation().getFileName()
         + "_" + thisJoinPoint.getSourceLocation().getLine())){

         try{
            String result = ThesisUtil.doXSSFix(param1,
               fixes.get(thisJoinPoint.getSourceLocation().getFileName()
               + "_" + thisJoinPoint.getSourceLocation().getLine()));
            return proceed(result);
         } catch (Exception e) {
            e.printStackTrace();
         }
      }
      else{
         return proceed(param1);
      }
      watch.stop();
      logger.info(thisJoinPoint.getSourceLocation().getFileName() + "_" +
         thisJoinPoint.getSourceLocation().getLine()
         + "," + watch.getElapsedTime());

      return proceed(param1);
   }

   pointcut xss_sendErrorFunction(String param1) :
      call(* *.sendError(String)) &&
      (within(*..BaseServlet) ||
      within(*..AVMWebappLoader) ||
      within(*..AVMRemoteTransportService) ||
      within(*..RepoRemoteTransportService) ||
      within(*..WebDAV) ||
      within(*..GetMethod) ||
      within(*..SampleEncryptionTransformer) ||
      within(*..TestDeploymentTransportTransformer)) &&
      args(param1);

   Object around(String param1) : xss_sendErrorFunction(param1){
      watch.start();
      if(fixes.containsKey(thisJoinPoint.getSourceLocation().getFileName()
         + "_" + thisJoinPoint.getSourceLocation().getLine())){

         try{
            String result = ThesisUtil.doXSSFix(param1,
            fixes.get(thisJoinPoint.getSourceLocation().getFileName()
            + "_" + thisJoinPoint.getSourceLocation().getLine()));
            return proceed(result);
         } catch (Exception e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
         }
      }
      else{
         return proceed(param1);
      }
      watch.stop();
      logger.info(thisJoinPoint.getSourceLocation().getFileName() + "_" +
         thisJoinPoint.getSourceLocation().getLine() + ","
         + watch.getElapsedTime());

      return proceed(param1);
   }

   pointcut xss_getParameterFunction(String param1) :
      call(* *.getParameter(String)) &&
      (within(*..BaseServlet) ||
      within(*..AVMWebappLoader) ||
      within(*..AVMRemoteTransportService) ||
      within(*..RepoRemoteTransportService) ||
      within(*..WebDAV) ||
      within(*..GetMethod) ||
      within(*..SampleEncryptionTransformer) ||
      within(*..TestDeploymentTransportTransformer)) &&
      args(param1);

   Object around(String param1) : xss_getParameterFunction(param1){
      watch.start();
      if(fixes.containsKey(thisJoinPoint.getSourceLocation().getFileName()
         + "_" + thisJoinPoint.getSourceLocation().getLine())){

         try{
            String result = ThesisUtil.doXSSFix(param1, fixes.get(
            thisJoinPoint.getSourceLocation().getFileName() + "_" +
            thisJoinPoint.getSourceLocation().getLine()));
            return proceed(result);
         } catch (Exception e) {
            e.printStackTrace();
```

```
            }
        }
        else{
            return proceed(param1);
        }
        watch.stop();
        logger.info(thisJoinPoint.getSourceLocation().getFileName() + "_" +
            thisJoinPoint.getSourceLocation().getLine() + ","
            + watch.getElapsedTime());

        return proceed(param1);
    }

    pointcut xss_setAttributeFunction(String param1) :
        call(* *.setAttribute(String)) &&
        (within(*..BaseServlet) ||
        within(*..AVMWebappLoader) ||
        within(*..AVMRemoteTransportService) ||
        within(*..RepoRemoteTransportService) ||
        within(*..WebDAV) ||
        within(*..GetMethod) ||
        within(*..SampleEncryptionTransformer) ||
        within(*..TestDeploymentTransportTransformer)) &&
        args(param1);

    Object around(String param1) : xss_setAttributeFunction(param1){
        watch.start();
        if(fixes.containsKey(thisJoinPoint.getSourceLocation().getFileName()
            + "_" +
            thisJoinPoint.getSourceLocation().getLine())){

            try{
                String result = ThesisUtil.doXSSFix(param1,
                        fixes.get(
                thisJoinPoint.getSourceLocation().getFileName()
                + "_" +
                thisJoinPoint.getSourceLocation().getLine()));
                return proceed(result);
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
        else{
            return proceed(param1);
        }
        watch.stop();
        logger.info(thisJoinPoint.getSourceLocation().
                getFileName()
            + "_" + thisJoinPoint.getSourceLocation().
                getLine()
            + "," + watch.getElapsedTime());

        return proceed(param1);
    }

    pointcut xss_writeByteArrayThreeParam(byte[] param1,
            String param2, String param3) :
```

```
      call(* *.write(byte[], String, String)) &&
      (within(*..BaseServlet) ||
      within(*..AVMWebappLoader) ||
      within(*..AVMRemoteTransportService) ||
      within(*..RepoRemoteTransportService) ||
      within(*..WebDAV) ||
      within(*..GetMethod) ||
      within(*..SampleEncryptionTransformer) ||
      within(*..TestDeploymentTransportTransformer)) &&
      args(param1, param2, param3);

   Object around(byte[] param1, String param2, String param3) :
      xss_writeByteArrayThreeParam(param1, param2, param3){
      watch.start();
      if(fixes.containsKey(
         thisJoinPoint.getSourceLocation().getFileName()
         + "_" + thisJoinPoint.getSourceLocation().getLine())){

         try{
            byte[] result = ThesisUtil.doXSSFix(param1,
               fixes.get(
            thisJoinPoint.getSourceLocation().getFileName()
            + "_" +
            thisJoinPoint.getSourceLocation().getLine()));
            return proceed(result, param2, param3);
         } catch (Exception e) {
            e.printStackTrace();
         }
      }
      else{
         return proceed(param1, param2, param3);
      }
      watch.stop();
      logger.info(thisJoinPoint.getSourceLocation().
         getFileName() + "_" +
         thisJoinPoint.getSourceLocation().
         getLine() + ","
            + watch.getElapsedTime());

         return proceed(param1, param2, param3);
      }
}
```

Appendix D SQL Injection Aspect Generated

```
package com.aspects;
import com.thesis.aop.util.ThesisUtil;
import com.thesis.aop.util.StopWatch;
import org.apache.log4j.Level;
import org.apache.log4j.Logger;
import org.apache.log4j.PropertyConfigurator;
import java.util.HashMap;
import org.owasp.esapi.errors.IntrusionException;
import org.owasp.esapi.errors.ValidationException;
import net.sf.jsqlparser.JSQLParserException;
```

```
public aspect SQLInjectionAspect{
   public HashMap<String, String> fixes =
      new HashMap<String, String>();
   Logger logger;
   StopWatch watch;

   public SQLInjectionAspect(){
      PropertyConfigurator.configure("log4j.properties");
      logger = Logger.getLogger("sqlInjectionLogger");
      logger.info("SQL Injection Aspect Created");
      watch = new StopWatch();

      fixes.put("SchemaBootstrap.java_1896", "SQL-ENCODE-MYSQL");
      fixes.put("SchemaBootstrap.java_605", "SQL-ENCODE-ORACLE");
      fixes.put("SchemaBootstrap.java_1043", "SQL-ENCODE-ORACLE");
      fixes.put("SchemaBootstrap.java_1876", "SQL-ENCODE-MYSQL");
      fixes.put("SchemaBootstrap.java_1917", "SQL-ENCODE-MYSQL");
      fixes.put("SchemaBootstrap.java_1271", "SQL-ENCODE-ORACLE");
      fixes.put("SchemaBootstrap.java_1883", "SQL-ENCODE-ORACLE");
      fixes.put("SchemaBootstrap.java_1931", "SQL-ENCODE-ORACLE");
      fixes.put("SchemaBootstrap.java_1890", "SQL-ENCODE-MYSQL");
      fixes.put("SchemaBootstrap.java_1266", "SQL-ENCODE-MYSQL");
      fixes.put("SchemaBootstrap.java_1910", "SQL-ENCODE-MYSQL");
      fixes.put("SchemaBootstrap.java_1903", "SQL-ENCODE-MYSQL");
      fixes.put("SchemaBootstrap.java_1924", "SQL-ENCODE-ORACLE");
   }

   pointcut sqlInjection_createSQLQueryFunction(String param1) :
      call(* *.createSQLQuery(String)) &&
      (within(*..SchemaBootstrap)) &&
      args(param1);

   Object around(String param1) :
      sqlInjection_createSQLQueryFunction(param1){
      watch.start();
      if(fixes.containsKey(thisJoinPoint.getSourceLocation().getFileName()
         + "_" +
         thisJoinPoint.getSourceLocation().getLine())){
         try{
            String result = ThesisUtil.doSQLInjectionFix(logger,
               param1, fixes.get(
               thisJoinPoint.getSourceLocation().getFileName() + "_" +
               thisJoinPoint.getSourceLocation().getLine()));
            if(result.contains("VALIDATION_FAILURE")){
               logger.info(result);
            }
            else{
               return proceed(result);
            }
         } catch (Exception e) {
            e.printStackTrace();
         }
      }
      else{
         return proceed(param1);
      }
      watch.stop();
      logger.info(thisJoinPoint.getSourceLocation().getFileName()
         + "_" +
         thisJoinPoint.getSourceLocation().getLine() +
         "," + watch.getElapsedTime());
      return proceed(param1);
   }

   pointcut sqlInjection_executeUpdateFunction(String param1) :
      call(* *.executeUpdate(String)) &&
      (within(*..SchemaBootstrap)) &&
      args(param1);

   Object around(String param1) :
      sqlInjection_executeUpdateFunction(param1){
      watch.start();
      if(fixes.containsKey(thisJoinPoint.getSourceLocation().
         getFileName() + "_" +
         thisJoinPoint.getSourceLocation().getLine())){
         try{
            String result = ThesisUtil.doSQLInjectionFix(logger,
               param1, fixes.get(thisJoinPoint.getSourceLocation().
                     getFileName() + "_" +
               thisJoinPoint.getSourceLocation().getLine()));
            if(result.contains("VALIDATION_FAILURE")){
               logger.info(result);
            }
            else{
               return proceed(result);
            }
         } catch (Exception e) {
            e.printStackTrace();
         }
      }
      else{
         return proceed(param1);
      }
      watch.stop();
      logger.info(thisJoinPoint.getSourceLocation().getFileName()
         + "_" + thisJoinPoint.getSourceLocation().getLine()
         + "," + watch.getElapsedTime());
      return proceed(param1);
   }

   pointcut sqlInjection_executeQueryFunction(String param1) :
      call(* *.executeQuery(String)) &&
      (within(*..SchemaBootstrap)) &&
      args(param1);

   Object around(String param1) :
      sqlInjection_executeQueryFunction(param1){
      watch.start();
      if(fixes.containsKey(thisJoinPoint.getSourceLocation().
         getFileName() + "_" +
         thisJoinPoint.getSourceLocation().getLine())){
         try{
            String result = ThesisUtil.doSQLInjectionFix(logger,
            param1, fixes.get(thisJoinPoint.getSourceLocation().
            getFileName() + "_" +
            thisJoinPoint.getSourceLocation().getLine()));
         if(result.contains("VALIDATION_FAILURE")){
            logger.info(result);
         }
         else{
            return proceed(result);
         }
      } catch (Exception e) {
         e.printStackTrace();
      }
   }
   else{
      return proceed(param1);
   }
   watch.stop();
   logger.info(thisJoinPoint.getSourceLocation().getFileName()
      + "_" + thisJoinPoint.getSourceLocation().getLine()
      + "," + watch.getElapsedTime());
   return proceed(param1);
}

pointcut sqlInjection_createQueryFunction(String param1) :
   call(* *.createQuery(String)) &&
   (within(*..SchemaBootstrap)) &&
   args(param1);

Object around(String param1) :
   sqlInjection_createQueryFunction(param1){
   watch.start();
   if(fixes.containsKey(thisJoinPoint.getSourceLocation().getFileName()
      + "_" + thisJoinPoint.getSourceLocation().getLine())){
      try{
         String result = ThesisUtil.doSQLInjectionFix(logger,
         param1, fixes.get(thisJoinPoint.getSourceLocation().
         getFileName() + "_" +
         thisJoinPoint.getSourceLocation().getLine()));
         if(result.contains("VALIDATION_FAILURE")){
            logger.info(result);
         }
         else{
            return proceed(result);
         }
      } catch (Exception e) {
         e.printStackTrace();
      }
   }
   else{
      return proceed(param1);
   }
   watch.stop();
   logger.info(thisJoinPoint.getSourceLocation().getFileName()
   + "_" + thisJoinPoint.getSourceLocation().getLine()
   + "," + watch.getElapsedTime());
   return proceed(param1);
}
```

```
pointcut sqlInjection_prepareStatementFunction(String param1) :
   call(* *.prepareStatement(String)) &&
   (within(*..SchemaBootstrap)) &&
   args(param1);

Object around(String param1) :
   sqlInjection_prepareStatementFunction(param1){
   watch.start();
   if(fixes.containsKey(thisJoinPoint.getSourceLocation().getFileName()
      + "_" + thisJoinPoint.getSourceLocation().getLine())){
      try{
         String result = ThesisUtil.doSQLInjectionFix(logger,
         param1, fixes.get(thisJoinPoint.getSourceLocation().
         getFileName() + "_" +
         thisJoinPoint.getSourceLocation().getLine()));
         if(result.contains("VALIDATION_FAILURE")){
            logger.info(result);
         }
         else{
            return proceed(result);
         }
      } catch (Exception e) {
         e.printStackTrace();
      }
   }
   else{
      return proceed(param1);
   }
   watch.stop();
   logger.info(thisJoinPoint.getSourceLocation().getFileName()
      + "_" + thisJoinPoint.getSourceLocation().getLine()
      + "," + watch.getElapsedTime());
   return proceed(param1);
}

pointcut sqlInjection_prepareStatementFunctionMultiParam(
   String param1, int param2, int param3) :
   call(* *.prepareStatement(String, int, int)) &&
   (within(*..SchemaBootstrap)) &&
   args(param1, param2, param3);

Object around(String param1, int param2, int param3) :
   sqlInjection_prepareStatementFunctionMultiParam(param1,
      param2, param3){
   watch.start();
   if(fixes.containsKey(thisJoinPoint.getSourceLocation().
      getFileName() + "_" +
      thisJoinPoint.getSourceLocation().getLine())){
      try{
         String result = ThesisUtil.doSQLInjectionFix(logger,
            param1, fixes.get(thisJoinPoint.getSourceLocation().
            getFileName() + "_" +
            thisJoinPoint.getSourceLocation().getLine()));
         if(result.contains("VALIDATION_FAILURE")){
            logger.info(result);
         }
```

```
            else{
               return proceed(result, param2, param3);
            }
         } catch (Exception e) {
            e.printStackTrace();
         }
      }
      else{
         return proceed(param1, param2, param3);
      }
      watch.stop();
      logger.info(thisJoinPoint.getSourceLocation().
         getFileName() + "_" +
         thisJoinPoint.getSourceLocation().getLine() + "," +
         watch.getElapsedTime());
      return proceed(param1, param2, param3);
   }

   pointcut sqlInjection_prepareStatementFunctionTwoParamIntArray(
      String param1, int param2) :
      call(* *.prepareStatement(String, int)) &&
      (within(*..SchemaBootstrap)) &&
      args(param1, param2);

   Object around(String param1, int param2) :
      sqlInjection_prepareStatementFunctionTwoParamIntArray(param1,
         param2){
      watch.start();
      if(fixes.containsKey(thisJoinPoint.getSourceLocation().getFileName()
         + "_" + thisJoinPoint.getSourceLocation().getLine())){
         try{
            String result = ThesisUtil.doSQLInjectionFix(logger,
            param1, fixes.get(thisJoinPoint.getSourceLocation().
            getFileName() + "_" +
            thisJoinPoint.getSourceLocation().getLine()));
            if(result.contains("VALIDATION_FAILURE")){
               logger.info(result);
            }
            else{
               return proceed(result, param2);
            }
         } catch (Exception e) {
            e.printStackTrace();
         }
      }
      else{
         return proceed(param1, param2);
      }
      watch.stop();
      logger.info(thisJoinPoint.getSourceLocation().getFileName()
         + "_" +
         thisJoinPoint.getSourceLocation().getLine() + ","
         + watch.getElapsedTime());
      return proceed(param1, param2);
   }

   pointcut sqlInjection_prepareStatementFunctionTwoParam(String param1,
      int[] param2) :
      call(* *.prepareStatement(String, int[])) &&
      (within(*..SchemaBootstrap)) &&
      args(param1, param2);

   Object around(String param1, int[] param2) :
      sqlInjection_prepareStatementFunctionTwoParam(param1, param2){
      watch.start();
      if(fixes.containsKey(thisJoinPoint.getSourceLocation().getFileName()
         + "_" +
         thisJoinPoint.getSourceLocation().getLine())){
         try{
            String result = ThesisUtil.doSQLInjectionFix(logger,
            param1, fixes.get(thisJoinPoint.getSourceLocation().
            getFileName() + "_" +
            thisJoinPoint.getSourceLocation().getLine()));
            if(result.contains("VALIDATION_FAILURE")){
               logger.info(result);
            }
            else{
               return proceed(result, param2);
            }
         } catch (Exception e) {
            e.printStackTrace();
         }
      }
      else{
         return proceed(param1, param2);
      }
      watch.stop();
      logger.info(thisJoinPoint.getSourceLocation().getFileName()
         + "_" + thisJoinPoint.getSourceLocation().getLine()
         + "," + watch.getElapsedTime());
      return proceed(param1);
   }

   pointcut sqlInjection_generalExecuteFunction(String param1) :
      call(* *.execute(String)) &&
      (within(*..SchemaBootstrap)) &&
      args(param1);

   Object around(String param1) :
sqlInjection_generalExecuteFunction(param1){
      watch.start();
      if(fixes.containsKey(thisJoinPoint.getSourceLocation().getFileName()
         + "_" + thisJoinPoint.getSourceLocation().getLine())){
         try{
            String result = ThesisUtil.doSQLInjectionFix(logger,
            param1, fixes.get(thisJoinPoint.getSourceLocation().
            getFileName() + "_" +
            thisJoinPoint.getSourceLocation().getLine()));
            if(result.contains("VALIDATION_FAILURE")){
               logger.info(result);
            }
            else{
               return proceed(result);
            }
         } catch (Exception e) {
            e.printStackTrace();
         }
      }
      else{
         return proceed(param1);
      }
      watch.stop();
      logger.info(thisJoinPoint.getSourceLocation().getFileName()
      + "_" + thisJoinPoint.getSourceLocation().getLine() + ","
      + watch.getElapsedTime());
      return proceed(param1);
   }
}
```

Appendix E XSS Validator Class

```
package com.thesis.aop.esapi;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

import org.owasp.esapi.ESAPI;
import org.owasp.esapi.ValidationRule;
import org.owasp.esapi.Validator;
import org.owasp.esapi.codecs.MySQLCodec;
import org.owasp.esapi.codecs.OracleCodec;
import org.owasp.esapi.errors.EncodingException;
import org.owasp.esapi.errors.IntrusionException;
import org.owasp.esapi.errors.ValidationException;
import org.owasp.esapi.reference.DefaultSecurityConfiguration;

public abstract class XSSValidation {

      public static String REGEX_ALPHANUMERIC = "AlphaNumberic";
      public static String REGEX_ALPHA = "Alpha";
      public static String REGEX_NUMERIC = "Numeric";
      public static String REGEX_EMAIL = "Email";
      public static String REGEX_ZIP_CODE = "ZipCode";
      public static String REGEX_IP_ADDRESS = "IPAddress";
      public static String REGEX_SSN = "SSN";

      public XSSValidation(){}

      public static String escapeCustomString(String s,
            String regex, int maxLength){

            Pattern p = ESAPI.securityConfiguration().
                  getValidationPattern(regex);

            if(p == null)
                  p = Pattern.compile(regex);

            //Sending pattern directly.
            try {
                  return ESAPI.validator().getValidInput(
                        "CUSTOM_STRING_ESCAPE",
                    s, p, maxLength, false, false);
            } catch (Exception e) {
                e.printStackTrace();
                String resultString = "";
                Matcher m = p.matcher(s);
                while(m.find()){
                    resultString += m.group(0);
                }
                return resultString;
            }
        }

        public static String escapeJavaScript(String s){
            return ESAPI.encoder().encodeForJavaScript(s);
        }

        public static String escapeCSS(String s){
            return ESAPI.encoder().encodeForCSS(s);
        }

        public static String escapeHTML(String s){
            return ESAPI.encoder().encodeForHTML(s);
        }

        public static String escapeHTMLAttribute(String s){
            return ESAPI.encoder().encodeForHTMLAttribute(s);
        }

        public static String escapeURL(String s) throws
            EncodingException{
            return ESAPI.encoder().encodeForURL(s);
        }

        public static String validateCreditCard(String s){
            try {
                s = ESAPI.validator().
                    getValidCreditCard("CREDIT_CARD", s, false);
            } catch (ValidationException e) {
                s = "VALIDATION FAILED " + s;
                e.printStackTrace();
            } catch (Exception e) {
                s = "VALIDATION FAILED " + s;
                e.printStackTrace();
            }
            finally{
                return s;
            }

        }
}
```

Appendix F SQL Injection Validator Class

```
package com.thesis.aop.esapi;

import java.io.IOException;
import java.io.InputStream;
import java.util.Iterator;
import java.util.List;
import java.util.Properties;
import java.util.regex.Matcher;
import java.util.regex.Pattern;
import javax.script.ScriptEngine;
import javax.script.ScriptEngineManager;
import javax.script.ScriptException;
import net.sf.jsqlparser.JSQLParserException;
import org.apache.log4j.Logger;
import org.owasp.esapi.ESAPI;
import org.owasp.esapi.codecs.MySQLCodec;
import org.owasp.esapi.codecs.OracleCodec;
import com.thesis.aop.main.Main;
import com.thesis.aop.sqlinjection.logic.SQLParser;
import com.thesis.aop.sqlinjection.parser.CONSTANT;
import com.thesis.aop.sqlinjection.parser.SimpleExpression;

public abstract class SQLInjectionValidation {

   private static Properties regexProperties;

   public static void loadRegexProperties() throws IOException{
      InputStream logStream = Main.class.
         getResourceAsStream("/properties/regex.properties");
      regexProperties = new Properties();
      regexProperties.load(logStream);
   }

   public static String escapeMySQL(String s, Logger logger)
         throws JSQLParserException {
      MySQLCodec mysql = new ~~MySQLCodec~~(MySQLCodec.MYSQL_MODE);
      String encodedResult = s;
      try {
         loadRegexProperties();
         encodedResult = removeCommentsFromString(
            regexProperties.getProperty("ALL_COMMENTS"),
            encodedResult, logger);
         encodedResult = removeCommentsFromString(
            regexProperties.getProperty("START_COMMENTS"),
            encodedResult, logger);

         List<SimpleExpression> items =
            SQLParser.getQueryValues(encodedResult);
         for (Iterator iterator = items.iterator(); iterator.hasNext();) {
            SimpleExpression simpleExpression = (SimpleExpression) iterator
                  .next();
            boolean tautology = testTautology(simpleExpression, logger);
            if (tautology) {
               logger.info("Obvious Tautology Detected with Left Side: "
                  + simpleExpression.getColumnName()
                  + " and Right Side: " + simpleExpression.value);
               logger.info("Replacing Obvious Tautology With '1=2' in " +
                     "order to prevent execution of attack");
               String regex = "(.{0,1}" + simpleExpression.columnName
                     + ".{0,1}\\s{0,1000}" + simpleExpression.op +
                     "\\s{0,1000}.{0,1}"
                     + simpleExpression.value + ".{0,1})";
               encodedResult = encodedResult.replaceAll(regex, " 1=2");
            } else if (simpleExpression.valueType == CONSTANT.VALUE_STRING) {
               encodedResult = encodedResult.replace(
                     "\"" + simpleExpression.value + "\"",
                     "\""
                           + ESAPI.encoder().encodeForSQL(mysql,
                                 simpleExpression.value) + "\"");
               encodedResult = encodedResult.replace(
                     "'" + simpleExpression.value + "'",
                     "'"
                     + ESAPI.encoder().encodeForSQL(mysql,
                     simpleExpression.value) + "'");
            }
         }
      } catch (JSQLParserException e) {
         logger.info("ERROR - INVALID QUERY: " + e.getCause());
         encodedResult = "PARSE_ERROR -" + encodedResult;
      } catch (IOException e) {
         logger.info("ERROR - IO Exception: " + e.getCause());
      }

      return encodedResult;

   }

   public static String escapeOracle(String s, Logger logger)
         throws JSQLParserException {
      OracleCodec oracle = new OracleCodec();
      String encodedResult = s;

      try {
         loadRegexProperties();
         encodedResult = removeCommentsFromString(
            regexProperties.getProperty("ALL_COMMENTS"),
            encodedResult, logger);
         encodedResult = removeCommentsFromString(
            regexProperties.getProperty("START_COMMENTS"),
            encodedResult, logger);

         List<SimpleExpression> items =
            SQLParser.getQueryValues(encodedResult);
         for (Iterator iterator = items.iterator(); iterator.hasNext();) {
            SimpleExpression simpleExpression = (SimpleExpression) iterator
                  .next();
            boolean tautology = testTautology(simpleExpression, logger);
            if (tautology) {
               logger.info("Obvious Tautology Detected with Left Side: "
                  + simpleExpression.getColumnName()
                  + " and Right Side: " + simpleExpression.value);
               logger.info("Replacing Obvious Tautology With '1=2' in " +
                  "order to prevent execution of attack");
               String regex = "(.{0,1}" + simpleExpression.columnName
                     + ".{0,1}\\s{0,1000}" + simpleExpression.op +
                     "\\s{0,1000}.{0,1}"
                     + simpleExpression.value + ".{0,1})";
            encodedResult = encodedResult.replaceAll(regex, " 1=2");
         } else if (simpleExpression.valueType == CONSTANT.VALUE_STRING) {
            encodedResult = encodedResult.replace(
                  "\"" + simpleExpression.value + "\"",
                  "\""
                  + ESAPI.encoder().encodeForSQL(oracle,
                     simpleExpression.value) + "\"");
            encodedResult = encodedResult.replace(
                  "'" + simpleExpression.value + "'",
                  "'" + ESAPI.encoder().encodeForSQL(oracle,
                  simpleExpression.value) + "'");
         }
      }
   } catch (JSQLParserException e) {
      logger.info("ERROR - INVALID QUERY: " + e.getCause());
      encodedResult = "PARSE_ERROR -" + encodedResult;
   } catch (IOException e) {
      logger.info("ERROR - IO Exception :" + e.getCause());
   }

   return encodedResult;
   }

   public static boolean testTautology(SimpleExpression exp,
   Logger logger) {
   ScriptEngineManager manager = new ScriptEngineManager();
   ScriptEngine engine = manager.getEngineByName("js");
   boolean resultBool = false;

   try {
      Object result;

      if (exp.getOp().equals("=")) {
         result = engine.eval("'" + exp.getColumnName() + "'" + "=="
               + "'" + exp.getValue() + "'");
      } else if (exp.getOp().equals("<>")) {
         result = engine.eval("'" + exp.getColumnName() + "'" + "!="
               + "'" + exp.getValue() + "'");
      } else {
         result = engine.eval(exp.getColumnName() + exp.getOp()
               + exp.getValue());
      }

      result = (Boolean) result;

      Boolean boolResult = new Boolean(result.toString());

      System.out.println(result);
      resultBool = boolResult.booleanValue();

   } catch (ScriptException e) {
      logger.info("ERROR in Tautology Detection", e);
   } finally {
      return resultBool;
   }

   }
```

```
   public static String removeCommentsFromString(String regex,
      String input, Logger logger){
      Pattern regexp = Pattern.compile(regex,
            Pattern.DOTALL | Pattern.MULTILINE);
       Matcher regexMatcher = regexp.matcher(input);
       while (regexMatcher.find()) {
          logger.info("Removing comment from query: \"" +
             regexMatcher.group() + "\" from \"" + input + "\"");
          input = input.replace(regexMatcher.group(), " ");
       }
       return input;
   }

}
```

References

1. “Cyber Security Statistics.” Entrepreneur Media Inc., n.p. Web. 13 Mar. 2006.
2. “Measuring Website Security: Windows of Exposure.” WhiteHat Website Security Statistic Report.,14 Mar. 2011.
3. Webroot. State of Internet Security – Protecting Enterprise Systems [Whitepaper] USA: Webroot Software Inc., 2007.
4. Electronista. “LulzSec hacks Sony Pictures, reveals 1m passwords unguarded.” Electronista Media Inc., 2 Jun. 2011.
5. Sethi, Rohit. “Aspect-Oriented Programming and Security”. SecurityFocus.com. 2 Nov. 2010.
6. Bostrom, G. Database Encryption as an Aspect. Proceedings of AOSD 2004 Workshop on AOSD Technology for Application level Security., Mar. 2004.
7. 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
8. 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.
9. Hermosillo, G., Gomez, R., Seinturier, L., Duchien, L. Using Aspect Programming to Secure Web Applications. Journal of Software, Vol. 2, No. 6., Dec 2007.
10. V. Shanmughaneethi, Ra. Yagna Pravin, C. Emilin Shyni, S. Swamynathan: SQLIVD - AOP: Preventing SQL Injection Vulnerabilities Using Aspect Oriented Programming through Web Services. HPAGC 2011: 327-337.
11. Clarke, Justin. SQL Injection Attacks and Defense. 1st ed. Syngress, 13 May 2009. 1 Mar. 2011.
12. Luong, Varian, "Intrusion Detection And Prevention System: SQL-Injection Attacks". *Master's Projects.* Paper 16. http://scholarworks.sjsu.edu., 12 Jan. 2010.
13. W. R. Cook, S. Rai, Safe Query Objects: Statically Typed Objects as Remotely Executable Queries, ICSE 2005
14. R. McClure, I. Kruger, SQL DOM: Compile Time Checking of Dynamic SQL Statements, ICSE 2005

15. V. B. Livshits, M. S. Lam, Finding Security Vulnerabilities in Java Applications with Static Analysis, USENIX Security Symp., 2005
16. G. Wassermann, Z. Su, *An Analysis Framework for Security in Web Applications*, pp. 70-78, SAVCBS 2004
17. Y. Huang et. al., *Securing Web Application Code by Static Analysis & Runtime Protection*, WWW2004
18. G. T. Buehrer, B. W. Weide, P. A. G. Sivilotti, *Using Parse Tree Validation to Prevent SQL Injection Attacks*, SEM 2005
19. W. G. Halfond, A. Orso, *Combining Static Analysis & Runtime Monitoring to Counter SQL-Injection Attacks*, WODA 2005
20. W. G. Halfond, A. Orso, *AMNESIA: Analysis and Monitoring for NEutralizing SQL Injection Attacks*, ASE 2005
21. OWASP (Open Source Web Application Security Project) . OWASP Top 10 – 2010 Edition. OWASP Foundation, 2010.
22. Klein, Amit. “DOM Based Cross Site Scripting or XSS of the Third Kind”. Web Application Security Consortium., 4 Jul. 2005.
23. Mece, Elinda. Kodra, Lorena. Towards full protection of Web Applications based on Aspect Oriented Programming, pp. 33-37, GJCST 2012.
24. Arthur, Charles. “Twitter users including Sarah Brown hit by malicious hacker attack.” Guardian News. 21 Sep. 2010.
25. Constantin, Lucian. “XSS Flaw Found on Secure American Express Site.” SoftPedia News. 5 Oct. 2010.
26. SpiderLabs. The Web Hacking Incident Database Semiannual Report. TrustWave Holdings Inc., 2011
27. D. Alhadidi, N. Belblidia, and M. Debbabi. AspectJ Assessment from a Security Perspective. In the Proceedings of the Workshop on Practice and Theory of IT Security, PTITS'2006, May 17-19, 2006.
28. Win, Bart ., Viren Shah, Wouter Joosen, and Ron Bodkin, editors. *AOSDSEC: AOSD Technology for Application-Level Security*, March 2004.

29. Bodkin, Ron. "Enterprise Security Aspects." *AOSDSEC: AOSD Technology for Application-Level Security*. Ed. Bart . Win, Viren Shah, Wouter Joosen, and Ron Bodkin, March 2004.
30. Fortify. Leading Bank Turns Security into a Differentiator with Fortify SCA. Fortify Software Inc. 2008.
31. SIIA. 2011 CODiE Awards Winners List. Software & Information Industry Association., 2011.
32. Laddad, Ramnivas. "AOP @ Work: AOP Myths & Realities." IBM Developer Works, 14 Feb. 2006.
33. Kersten, Mik. "AOP @ Work: AOP Tools Comparison Part 2." IBM Developer Works, 8 Feb. 2005.
34. Miles, Russ. *AspectJ Cookbook*. 1st ed. O'Reilly Media, 2004. Print.
35. Almaer, Dion. "AspectJ and AspectWerkz Merge Forces." The Server Side. 19 Jan 2005.
36. Alfresco Community. Alfresco Content Management. Alfresco Software Inc. 31 Jan. 2012. Web.
37. JadaSite. JadaSite E-Commerce System. JadaSite.com, 2011. Web.
38. Partho. "10 Best Open Source ERP Software." Gaea News Network, 27 Oct. 2009.
39. Doyle, Maureen. Walden, James. An Empirical Study of the Evolution of PHP Web Application Security, pp. 11-20, METRISEC 21 Sept. 2011.
40. Chess, Brian, and Jacob West. *Secure Programming With Static Analysis*. 1st ed. Addison-Wesley Professional, 2007. Print.
41. Manico, Jim. OWASP – Enterprise Security API. OWASP Foundation, 31 Mar. 2010.
42. Alfresco Community. Alfresco Customer Overview. Alfresco Software Inc, 2012. Web.
43. "Fortify Source Code Analyzer – Capabilities." HP Fortify. Web. 2011.
44. DUPLICATE OF 21
45. Houston, Pete. Android XML Adventure – Parsing XML Data with SAXParser. Web. 9 Oct. 2011.
46. AspectJ Programming Guide – Pointcut Language Semantics. Eclipse Foundation. Web. 2012.

47. AspectJ Programming Guide – Advice Language Semantics. Eclipse Foundation. Web. 2012.
48. AspectJ Programming Guide – Chapter 2 Join Points. Eclipse Foundation. Web. 2012.
49. Brereton, JoAnn. Use JavaCC to build a user friendly Boolean query language. IBM Developer Works. Web. 15 Jan. 2004.
50. Tamada, Srinivas. Hack your Own Web Project: SQL Injection. 9Lessons.info. Web. 15 Dec. 2008.
51. Wiegenstein, Andreas. The Imact of Cross Site Scripting on Your Business. Virtual Forge. 27 Jul. 2007.
52. ESAPI Interface Encoder. The Open Web Application Security Project. Web. 2011.
53. ESAPI Validator Library. The Open Web Application Security Project. Web. 2011.
54. Li, Sing. AOP: Patching in the 21st Century. Developer Fusion. Web. 23 Jul. 2010.
55. RSnake. XSS (Cross Site Scripting) Cheat Sheet Esp: for filter evasion. Ha.ckers. Web. 2011.