Protection of Web Applications Using Aspect Oriented Programming and Performance Evaluation

Elinda Kajo-Mece
Faculty of Information Technology
Polytechnic University of Tirana
Albania
ekajo@fie.upt.al

Lorena Kodra
Faculty of Information Technology
Polytechnic University of Tirana
Albania
lorena.kodra@gmail.com

Enid Vrenozaj
Faculty of Information Technology
Polytechnic University of Tirana
Albania
enidv11@gmail.com

Bojken Shehu
Faculty of Information Technology
Polytechnic University of Tirana
Albania
b.shehu@fti.edu.al

ABSTRACT
Web application security is a critical issue. Security concerns are often scattered through different parts of the system. Aspect oriented programming is a programming paradigm that provides explicit mechanisms to modularize these concerns. In this paper we present an Aspect Oriented system for detecting and prevent common attacks in web applications like Cross Site Scripting (XSS) and SQL Injection and evaluate its performance by measuring the overhead introduced into the web application. The results of our tests show that this technique was effective in detecting attacks while maintaining a low performance overhead.

General Terms
Algorithms, Performance, Security

Keywords
Security, web application, XSS, SQL injection, aspect oriented programming, performance evaluation

1. INTRODUCTION
Today, user and critically important company information is managed using web applications. For this reason, web applications can be a door for attacks. The vulnerabilities present in the application can be exploited by an attacker. Even with the rapid development of Internet technologies, web applications have not achieved the desired security levels. As a result, web servers and web applications are popular attack targets.

The most common attacks on web applications are Cross Site Scripting (XSS) and SQL Injection [14]. SQL Injection is a technique where an attacker injects SQL code into the user input field in order to modify the original structure of the query to post hidden data, or execute arbitrary queries in the database. Cross Site Scripting occurs when an attacker injects and executes scripts written in languages like JavaScript or VBScript.

In the case of SQL injection a string is considered to be dangerous when it alters the original structure of the sql statement. These strings tipically contain non-alpha numeric characters such as <, >, =, –, ','' and/or a combination of the SQL keywords.

BCI'12, September 16–20, 2012, Novi Sad, Serbia.

Local Proceedings also appeared in ISBN 978-86-7031-200-5, Faculty of Sciences, University of Novi Sad.

In the case of XSS we consider a dangerous string the one containing non-alpha numeric characters such as <, >, =, –, ','".

Aspect Oriented Programming is a programming paradigm that provides explicit mechanisms to modularize crosscutting concerns (behavior that cuts across different divisions of the software) such as security. This means that the security code can be separated from the system code which makes it a good candidate for applying security to a system compared to traditional programming methodologies where the security code may be spread over multiple modules of the system and its maintenance would be difficult and error prone.

In this paper, we propose and evaluate an Aspect Oriented protection system that detects and prevents attacks on web applications. This system analyzes and validates user input strings. We use an aspect to capture input strings and compare them to predefined patterns. The intrusion detection aspect is implemented in AspectJ and is woven into the target system. The resulting system has the ability to detect malicious user input and prevent SQL Injection and Cross Site Scripting. The advantage in using aspect oriented programming lies in separating the security code from application code. In this way it can be developed independently to adapt to new attacks.

We test our system by applying it to an insecure web application and we evaluate its impact on the overall web application performance.

The rest of the paper is organized as follows. Section 2 presents concepts of SQL Injection, XSS and AOP. Section 3 describes our proposed solution. Section 4 describes in details the architecture of our system and its integration with the web application. Section 5 describes the experimentation and evaluation results. Section 6 concludes and discusses some future work.

2. CONCEPTS OF SQL INJECTION, XSS AND ASPECT ORIENTED PROGRAMMING
The basic idea behind SQL Injection is inserting malicious SQL commands into a parameter that a web application sends to a database. The malicious SQL commands alter the original intended structure of the query and it becomes a malicious query. If it is executed it may corrupt or even destroy the database. The most popular techniques used in SQL injection are tautology, union, and comments.

The general idea behind tautology is inserting malicious code into one or more conditional statements of a SELECT or UPDATE statement so that they always evaluate as true. Let's consider the case where the web application authenticates users by executing the following query:

```
SELECT * FROM users WHERE username = 'admin'
and password = 'pass'
```

This query doesn't select any rows because the password is incorrect. Injecting `'OR 1=1` gives:

```
SELECT * FROM users WHERE username = 'admin'
and password = ''OR 1=1'
```

This causes the WHERE clause to be true for every row and all table rows are returned.

An attacker can use the UNION clause to manipulate an SQL statement into returning rows from another table. Let's consider the following query that allows users to get the product name by inserting the product ID.

```
SELECT productName FROM products WHERE
productID = '3'
```

An attacker can use the UNION clause to modify the structure of this query to:

```
SELECT productName FROM products WHERE
productID = '3' UNION SELECT username,
password FROM users
```

This query will display the product name together with the usernames and passwords of the users table.

Another type of SQL Injection uses comments to change the structure of an SQL query. The part of the SQL statement that comes after the comments will not be executed and the query will return the results that the attacker wanted. Let's consider the following SQL statement:

`SELECT * FROM users WHERE username = 'alice' and password = 'alice*123'` can be transformed in the following way:

```
SELECT * FROM users WHERE username = 'admin'
-- and password = ''
```

The query will return all the information about the admin user.

Cross Site Scripting (XSS) is an attack done against the user's browser in order to attack the local machine, steal user information, spoof the user identity, attack other machines, etc. The attacker uses a web application to send malicious code usually in the form of a script. Together with the legitimate content, the users get the malicious script from the web application. This attack is successful in web applications that do not validate user input.

Aspect Oriented Programming is a programming paradigm that aims to solve problems like code scattering and code tangling that cannot be solved by traditional programming methodologies. Code scattering means that the problem code is spread over multiple modules. This makes things difficult for the developers when they want to fix a bug because they have to modify several source files. Code tangling means that the problem code is mixed with other code. In the case of web applications, security code needs to be applied in different modules of the system. This process is error prone and difficult to deal with. AOP is a good candidate for applying security in web applications. The security code can be encapsulated into modules called aspects which can be maintained separately from the web application in order to adapt to new attacks.

3. RELATED WORK AND PROPOSED SOLUTION

Different solutions have been proposed addressing security issues in web applications. Some of these solutions are presented below.

Zhu and Zulkerine propose a model-based aspect-oriented framework for building intrusion-aware software systems [2]. They model attack scenarios and intrusion detection aspects using an aspect-oriented Unified Modeling Language (UML) profile. Based on the UML model, the intrusion detection aspects are implemented and woven into the target system. The resulting target system has the ability to detect the intrusions automatically.

Mitropoulos and Spinellis propose a method for preventing SQL Injection attacks by placing a database driver proxy between the application and its underlying relational database management system [1]. To detect an attack, the driver uses stripped-down SQL queries and stack traces to create SQL statement signatures that are later used to distinguish between injected and legitimate queries. The driver depends neither on the application nor on the RDBMS.

Hermosillo et al. present "AProSec" implemented in AspectJ and in the JBoss AOP framework, a security aspect for detecting SQL Injection and XSS [3]. They use the same aspect for dealing with SQL Injection and XSS. Their experiments show the advantage of runtime platforms such as JBoss AOP for changing security policies at runtime.

The authors in [4] present Noxes, which is a client-side solution to mitigate cross-site scripting attacks. Noxes acts as a web proxy and uses both manual and automatically generated rules to mitigate possible cross-site scripting attempts. Analogous to personal firewalls, Noxes allows the user to create filter rules (i.e., firewall rules) for web requests.

In [5], Madow et al. describe a method for defending web applications against XSS vulnerabilities at runtime by identifying dangerous values as they are written into the HTTP response rather than as they enter the program. Their method is comprised of two phases: an attack-free training period where they capture the normal behavior of the application in the form of a set of likely program invariants, and an indefinite period of time spent in a potentially hostile environment where they check to make sure the application does not deviate from the normal behavior.

Janot, and Zavarsky in [6] present a study on the prevention of SQL Injections, an overview of proposed approaches and existing solutions, and recommendations on preventive coding techniques for Java-powered web applications and other environments. They present their solution, the SQLDOM4J, which targets Java environments and enables developers to construct and execute safe SQL statements easily and to protect applications against SQL Injection Attacks.

The solution we propose performs a two-step validation of user input. In the first step, the user input is validated syntactically by the Syntactic Validator to check whether it contains dangerous characters that can be used in XSS and SQL Injection. In the second step, the input is validated by the SQL validator (Semantic Validator) in the context of a query. This is done to check whether it contains always true statements or combinations of

SQL keywords that can modify the original structure of the query and make it dangerous. What makes our system different from the solutions described earlier is the fact that it analyzes directly user input before it is being used as part of an SQL query. This facilitates the analyzing process because the part of the query that is pre-programmed is considered safe and there is no need to validate it. Another advantage of our system is the fact that the SQL validator checks the presence of SQL keywords in the user input. This prevents attacks that do not contain comments or always true statements but contain SQL keywords that can modify the original structure of the SQL query. The presence of a combination of SQL keywords is considered as an attack, not the presence of a single SQL keyword. This prevents the generation of false positives in cases when for example the word "Union" is part of a legitimate user name.

3.1 System Architecture

Our system consists of three parts. The first and the most important part is an aspect implemented in AspectJ [10] called WebAppInputFilter that contains the logic of the whole defense process. It contains the advices that control the validation process as well as the actions to be taken (code to be executed) based on the results of the validation. The aspect also contains the pointcuts that define the vulnerable points of the web application and allow the weaving with the advice code. It monitors the traffic in servlets and captures some specific calls that implement the *ServletRequest* and *HttpServletRequest* interfaces. The pointcuts are:

```
pointcut pcGetParameter(): call(String
javax.servlet.http.HttpServletRequest.getPar
ameter (String))
pointcut pcGetParameterValues():call(String
[]
javax.servlet.ServletRequest.getParameterVal
ues(String))
pointcut pcGetQueryString(): call String
javax.servlet.http.HttpServletRequest.getQue
ryString (String))
pointcut pcGetRequestURL(): call(String
javax.servlet.http.HttpServletRequest.getReq
uestURL (String))
```

These pointcuts cover even the cases when the attacker performs the attack directly from the URL without the need for using a form.

The second part consists of a validators class that validate against XSS and SQL Injection attacks the input captured by the advices. The third part consists of an encoder which encodes dangerous characters by converting them to their decimal equivalent, making them harmless.

The basic idea behind our technique is to capture user input and validate it by comparing it to predefined patterns. This makes our system different from the ones described in [1, 2, 3, 4, 5, 6]. As we explained before, the user input is validated before being used as part of a query. The query that is entered into the web application is a combination of user input and a partial SQL statement defined by the developer. The part defined by the developer is considered as safe so there is no need to validate it and we only validate the user input part. This speeds up the validation process, makes it easier and also decreases the possibility of generating false positives.

The validation process happens in two steps. In the first step it is checked whether the input contains dangerous characters such as '<',' >', '=' and' –'that can be used to perform XSS and SQL Injection attacks. In the second step, the input is analyzed by the SQL Validator (Semantic Validator) in the context of the query. This is done to check whether the query contains combined SQL keywords that can modify the original structure of the query or SQL code that can transform the original query in an SQL statement that results always true.

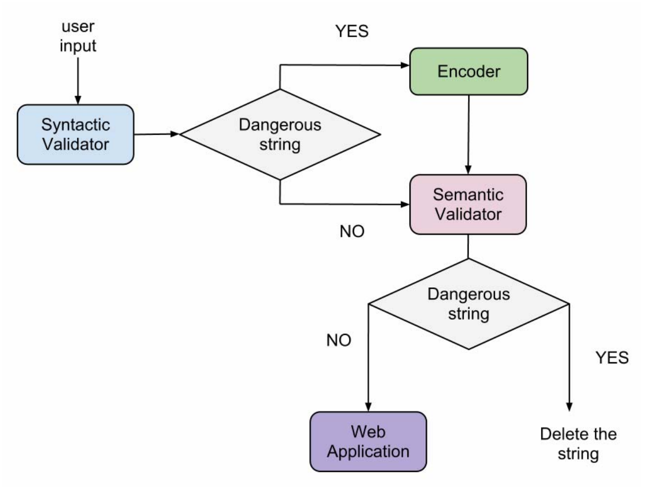

Figure 1: The flow of information within the defense system.

Figure 1 shows the flow of information within the defense system. The aspect captures the user input string and sends it to the first analyzer (Syntactic Validator). If the string is not dangerous it is passed on to the second validation step (Semantic Validator). If the string is dangerous it is send to the encoder. It encodes the dangerous characters and the result is passed to the Semantic Validator. If the string is not considered dangerous, it is passed on to the web application as a legitimate request. If it is considered dangerous, it is erased.

In the cases when an attack is detected the system generates a notification showing the dangerous string which can be saved on a log file or database for later analysis.

4. PERFORMANCE EVALUATION

We tested our system by using it to protect a vulnerable web application [11]. First we applied all sorts of SQL Injection and XSS injection attacks on the unprotected system to see how it behaved. Then we applied our solution to protect it and attacked it again but were unable to bypass the application's security. The results of these tests are described further in [7]

In order to evaluate the impact of the defense system on the performance of the web application we measured its response time using JMeter [12], first in the absence of the defense system and then in the presence of our defense system. JMeter is a tool used to test performance both on static and dynamic resources. It can be used to simulate a heavy load on a web application or to analyze overall performance under different load types. We created 3 scenarios with GET and POST HTTP requests:

- POST requests for XSS
- POST requests for SQL Injection

- A mix of legitimate requests, XSS attack requests and SQL Injection requests.

Each of these scenarios represent real life situations where the requests can be only XSS attacks, only SQL injection attacks or a mix of legitimate requests, XSS and SQL injection attacks. XSS and SQL Injection attacks use only POST requests since they try to insert information into the web application. GET requests represent the legitimate requests when the user wants to retrieve information from the web application. POST requests also are used as legitimate requests in cases where the user wants to insert information into the web application.

For every scenario we evaluated the system under different loads. We used 3 different requests loads: 1000 requests, 10000 requests and 100000 requests. These numbers of requests are used to simulate normal, high and very high load. The default maximum number of concurrent threads in the Tomcat's server configuration file is set to 150. We could spawn as many client threads as we wish but if we exceed the limit of 150, performance will decrease because some client request threads will always be waiting. So it is better to stay just under the maximum number, such as 149 client threads. We executed each test 5 times and measured the average response time and calculated the overhead introduced. The response time is the time the web application takes to process a user request and send the result to the user. The overhead is the difference in response times between the system with protection and without protection.

Table 1 shows the results for the first scenario where all the requests are harmful POST requests for XSS.

Table 1: Overhead for POST Requests for XSS

	1000 requests	**10000 requests**	**100000 requests**
Overhead (%)	2.14	2.21	2.31

As we can see, the defense system introduces an overhead. This overhead comes from the Syntactic Validator and the Encoder We also notice that with the increasing number of requests, the overhead increases too. This comes from the increased processing and encoding of the requests.

Table 2 shows the results for the second scenario where all the requests are harmful POST requests for SQL Injection.

Table 2: Overhead for POST Requests for SQL Injection

	1000 requests	**10000 requests**	**100000 requests**
Overhead (%)	1.99	2.10	2.23

As we can see from the results, even in this case the defense system introduces an overhead. If we compare these two tables we notice that the overhead is slightly smaller in the case where the requests are for SQL Injection. This comes from the fact that not all strings need to be encoded.

Table 3 shows the results for the third scenario where we have a mix of harmful and harmless requests for XSS and SQL Injection.

Table 3: Overhead for POST and GET Requests for Harmless and Harmful Requests for XSS and SQL Injection

	1000 requests	**10000 requests**	**100000 requests**
Overhead (%)	1.93	2.03	2.19

As we can see, the overhead introduced into the system in this case is lower due to the presence of harmless requests that don't need to be encoded.

We feel that the overhead introduced in the scenarios described above is at an acceptable level for use in many production environments and it will not be noticeable by the user. Hence we can conclude that our system offers an effective defense against XSS and SQL Injection by keeping at the same time a low performance overhead.

5. CONCLUSIONS AND FUTURE WORK

We have presented an approach for building an effective security system for a web application. This system detects XSS and SQL Injection attacks in requests. Our system was built separately and the initial code of the web application was not modified. This allows the security system to be evolved independently from the web application to adapt to new attacks.

What makes our system different from similar proposed solutions is the fact that it analyzes directly user input before it is being used as part of an SQL query. This facilitates and speeds up the analyzing process because the part of the query that is pre-programmed is considered safe and there is no need to validate it.

The performance evaluation under normal, moderate and high load showed that our defense system provides an effective protection while maintaining within acceptable levels the overhead introduced in the system.

Our system can be improved in some directions. A possible improvement might be the implementation of defense against other form of attacks. Also new techniques like machine learning and neural networks can be used to detect more sophisticated attacks. Another direction of improvement might be the implementation of runtime weaving using the JBoss AOP Framework [13].

6. REFERENCES

[1] Mitropoulos, D., Spinellis, D. Sdriver. 2009. Location-specific signatures prevent SQL injection attacks, Computers & Security, Vol.28, Issues 3-4, (May-June 2009,) 121-129

[2] Zhu, Z.J., Zulkernine, M. 2009.A model-based aspect-oriented framework for building intrusion-aware software systems, Information and Software Technology, Vol.51, Issue 5, (May 2009), 865-875

[3] Hermosillo, G., Gomez, R., Seinturier, L., Duchien, L. 2007. AProSec: An aspect for programming secure web applications, In Proceedings of the The Second International Conference on Availability, Reliability and Security, (2007), 1026-1033

[4] Kirda, E., Jovanovic, N., Kruegel, C., Vigna, G. 2009. Client-side cross-site scripting protection, Computers & Security, Vol.28, Issue 7, (October 2009), 592-604

[5] Madou, M., Lee, E., West, J., Chess, B. 2008. Watch What You Write: Preventing Cross-Site Scripting by Observing Program Output, OWASP AppSec 2008 Conference (AppSecEU08)

[6] Janot, E., Zavarsky, P. 2008. Preventing SQL Injections in Online Applications: Study, Recommendations and Java Solution Prototype Based on the SQL DOM, OWASP AppSec 2008 Conference (AppSecEU08)

[7] Kodra, L., Kajo, E. 2011. Protecting Web Applications using AspectJ . *6th Annual South East European Doctoral Student Conference*, (Thessaloniki, Greece, September 19-20, 2011).

[8] Lam, M.S., Martin, M., Livshits., Whaley, J. 2008. Securing Web Applications with Static and Dynamic Information Flow Tracking. *ACM SIGPLAN 2008 Workshop on Partial Evaluation and Program Manipulation* (San Francisco, USA, January 7-8, 2008). PEPM '08.

[9] Lam, M.S., Martin, M., 2008. Automatic generation of XSS and SQL injection attacks with goal-directed model checking. *In Proceedings of the 17th conference on Security symposium* (USENIX Association Berkeley, USA. 2008)

[10] AspectJ, http://www.eclipse.org/aspectj/

[11] WebGoat,http://www.owasp.org/index.php/Category:OWASP_WebGoat_Project.

[12] Apache Jmeter, http://jakarta.apache.org/jmeter/.

[13] JBoss AOP, http://www.jboss.org/jbossaop

[14] OWASP Top Ten Project, https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project