# AProSec: an Aspect for Programming Secure Web Applications

Gabriel Hermosillo, Roberto Gomez, Lionel Seinturier, Laurence Duchien

# AProSec: an Aspect for Programming Secure Web Applications

Gabriel Hermosillo - Roberto Gomez-Cardenas
ITESM-CEM-Dpto. Ciencias Computacionales
Km. 3.5 Carretera Lago de Guadalupe
52926 Edo de Mexico, Mexico
+53 (55) 58.64.56.45
ghermosillo@itesm.mx,rogomez@itesm.mx

Lionel Seinturier - Laurence Duchien
LIFL – INRIA Jacquard Project
Cité Scientifique – Bât M3
F-59655 Villeneuve d'Ascq – France
+33 (0) 3 28 77 85 84
Lionel.Seinturier@lifl.fr, Laurence.Duchien@lifl.fr

## ABSTRACT

Adding security functions in existing Web application servers is now vital for the IS of companies and organizations. Writing crosscutting functions in complex software should take advantage of the modularity offered by new software development approaches. With Aspect-Oriented Programming (AOP), separating concerns when designing an application fosters reuse, parameterization and maintenance. In this paper, we design a security aspect called AProSec for detecting SQL injection and Cross Scripting Site (XSS) that are common attacks in web servers. We experiment this aspect with the AspectJ language and the JBoss AOP framework. With this experimentation, we show the advantage of runtime platforms such as JBoss AOP for changing security policies at runtime. Finally, we describe related work on security and AOP.

## 1. INTRODUCTION

Companies and organizations use Web servers to publish information that concerns directly their users. However, other institutions consult their operations through these same servers. The ignorance of the developers concerning the vulnerabilities of this kind of systems, highlights the weakness of these software. OWASP's Top Ten listing references two common attacks on this type of systems: Cross Site Scripting (XSS) and SQL injection [1]. SQL injection is a technique where a would-be intruder modifies an existing SQL request to post hidden data, to crush important values, or to process dangerous orders for the database. That is made when the application retrieves data sent by the Internet users, and uses it directly to build a SQL request. Cross Site Scripting (XSS) is an attack exploiting a weakness of a Web site that fails to validate the parameters entered by the users. XSS uses various techniques for injecting (and executing), scripts written in languages such as JavaScript or VBScript. The goal of these attacks is to keep cookies containing information identifying users, or to mislead them later so that they provide these data to the attacker.

Security techniques used by most web developers do not perform very well. The approach *Design for security* defends the idea that security should be taken into account during all the phases of the development cycle and must influence deeply the design of the application.

Aspect-Oriented Programming (AOP) is a good candidate for this feature [2]. AOP has been proposed as a technique for improving separation of concerns in software systems and for adding crosscutting functionalities without changing the business part of the software. AOP provides specific language mechanisms that make possible to address concerns, such as security, in a modular way. AOP languages and tools can be applied at compile-time or at run-time. This way, the security issue in a software system can be addressed

Our main objective is to design and implement a security aspect called AProSec to deal with SQL Injection and XSS web attacks. Our proposal is based on the aspect programming models offered by AspectJ and JBoss AOP and defines the elements necessary for the defense of a Web site against these attacks, not only by validating and filtering the user info, but also by implementing a SQL analyzer that can intercept and validate all the database queries before they are processed. These elements will appear as AspectJ [3] aspects woven at compile-time and, in a second version, at run-time with the JBoss AOP [4] framework.

Our work is motivated by the need to fill the gap between an integrated version of a web server with security functions and a modular version with AOP techniques. This paper leads to the definition of a model for addressing security issues in software applications that could be re-used on several software systems with few changes and be dynamically added at the runtime.

The rest of this paper is organized as follows. Section 2 presents the motivation and principles of SQL Injection, XSS and AOP. Section 3 provides the Web application architecture. Section 4 defines our AProSec Aspect, its integration with the web server architecture and details the difference between two weaving with AspectJ and JBoss AOP. Section 5 describes some related work. Finally, Section 6 concludes and discusses some future work.

## 2. MOTIVATION AND PRINCIPLES

### 2.1 SQL injection and XSS

*SQL injection*

According to [1] a SQL injection attack consists in finding a parameter that a web application sends to a database. The attacker embeds malicious SQL commands into parameters in order to trick the web application for forwarding a malicious query to the database. As a result of this kind of attack, the database contents can be corrupted, destroyed or disclosed.

Many techniques are used in SQL injection. The most popular are tautology, union, additional declaration and comments. In order to

explain each technique, we will consider the case in which a web application authenticates a user by executing the following query:

```
SELECT * FROM users WHERE name='alice' and password = 'toto'
```

Tautology looks for a disjunction in the WHERE clause of a select or update statement. In the previous example it can be made by adding the statement 'a'='a', resulting in the following query:

```
SELECT * FROM users WHERE user='alice' and password = 'toto' or 'a' = 'a'
```

The precedence operator causes the `WHERE` clause to be true for every row, and all table rows will be returned.

The union clause allows grouping the result of two SQL queries. The goal is to manipulate a SQL statement into returning rows from another table. As an example we will assume that a database containing the reports is available:

```
SELECT body, results FROM reports
```

When using this statement with our example, we will obtain the following query:

```
SELECT body, results FROM reports
UNION
SELECT * FROM users
```

As result the query will display the reports list, but also the database users in the application.

The additional statements technique attempts to add SQL statements or commands to a SQL query. For example:

```
SELECT * FROM users WHERE name='alice' and password = 'toto'
DELETE FROM users WHERE username = 'admin'.
```

When executing the previous query, the admin record would be erased from the database.

We can also use comments. Most of the databases use the "--" or "#" characters for a comment indication. An attack can use the comments to cut a SQL query and change the meaning of it. For example the following SQL statement:

```
SELECT * FORM users WHERE name = 'alice' and password = 'toto'
```

can be transformed in the following way:

```
SELECT * FORM users WHERE name = 'admin' -- and password = ''
```

The result will show all the information about the admin user in the user's database. All these attacks can be combined to form more complex SQL queries.

### *XSS*

The XSS cross site scripting is an attack oriented to the user's browser, in order to disclose the end user's token, to attack the local machine, or to spoof content to fool the user [1]. The attacker uses a web application to send malicious code generally in the form of a script to a particular user. The attack takes advantage of web applications that do not validate the output generated by a user's input. The attack is known as XSS attack, and not CSS attack, to avoid confusion with Cascading Style Sheets.

As an example, consider a web application that gives the visiting user the opportunity to send a comment through a guest book. A malicious user can introduce the following characters "<! --". After some time, these characters are mixed with other users' input, resulting in the following content in the guess book:

```
Very good web page, dude!
<!--
You're da man, boss
```

When a user reads the guest book with a browser, it will read all the contents and will interpret the character "<!--" not as a user's opinion, but as a HMTL tag. As a result, the rest of the content in the guest book is ignored by the users' browsers. We can imagine the effects of the following statements in the guest book.

```
<script>
  for (q=0; q < 1000; q++)
        window.open(http://www.hot.example);
</script>
```

This is an example of a very simple XSS attack. An attacker can introduce scripts that can take session cookies of a user and send them to the attacker. With this information the attacker can use the system as the original user.

## 2.2 AOP

The domain of aspect-oriented programming (AOP) [2][3] appeared in 1996. It was pioneered by Gregor Kiczales and his team, then at the Xerox Palo Alto Research Center. While original and innovative, the domain of AOP inherits results from other programming approaches such as reflection, open implementations, meta-object protocols or generative programming.

One of the experiences that motivated the definition of AOP was the study of the Tomcat servlet engine. When studying the code of Tomcat, Gregor Kiczales and his team discovered that, while some functionality was cleanly modularized in classes, other, such as user session management or logging, appeared in several classes. This phenomenon is known as code scattering. When developers want to fix a bug or to upgrade such functionalities, they have to scan and modify several source files. While feasible, this hinders productivity and is error-prone. In other cases, the code scattered around several classes, was also redundant. The consequence of this scattering is that a given method mixes concerns related to different functionalities. This second phenomenon is known as code tangling. Once again this hinders the maintainability and understandability of applications.

When faced with these two phenomena, the question is whether scattering and tangling are irreducible or is the result of a poor design. In other words, could Tomcat be re-designed to prevent scattering and tangling? While open, the answer to this question is usually no. The idea is that a complex piece of software such as Tomcat may be decomposed according to many criteria: the decomposition may be data-driven, process-driven, driven by various requirements such as security, integration with existing information systems, or performance. It happens that one is chosen by designers and that the other decompositions may not fit in the scheme introduced by the first one, leading to functionalities being scattered and tangled. The purpose of AOP is then to provide a solution to solve these issues.

AOP, as a new programming paradigm, introduces notions such as an aspect, a join point, a pointcut and an advice code. However, these notions do not replace existing ones such as a class, an object, a procedure or a method. Rather, AOP must be seen as a complement to these existing techniques. Furthermore, these notions are not specific to a programming style (e.g. object-oriented or procedural) or a given syntax (Java, C#, Ada, COBOL, etc.). Aspect-oriented extensions exist for many languages, object-oriented or procedural. Furthermore, aspects can be applied (the term used by the AOP community is woven)

at compile-time or at run-time. Experience has shown the difficulty of writing crosscutting functions such as security [5].

# 3. Web application architecture

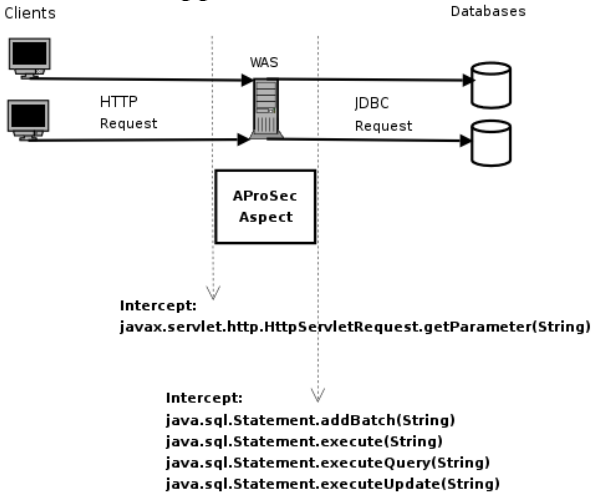


**Figure 1: The architecture of our Web application server**

Figure 1 shows the architecture of our Web Application Server (WAS). The client sends a request to the Web Application Server. This HTTP request is intercepted and validated by the AProSec aspect. If needed, the WAS sends a request to the Database in order to get a response for the client. This latter JDBC request is also intercepted and validated by the AProSec aspect. If the request is correct, it will be processed, otherwise it is rejected.

# 4. The AProSec Aspect

The AProSec aspect can be used by any AOP framework and is composed of three parts. First, an advice (the added code) defines the validation process. Second, the way AProSec validates the requests depends on the options that the administrator selects on the configuration file, as shown in Section 4.2. Finally, the pointcut part (where the code is added) allows the weaving with the WAS. How this weaving is made will be described in Section 4.3 and 4.4 for each implementation.

## 4.1 Advice

The advice part consists in two main validations:

1. HTTP requests parameters (intercepting javax.servlet.http. HttpServletRequest.getParameter(String) call),

2. DB queries (intercepting java.sql.Statement.addBatch(String), execute(String), executeQuery(String) and executeUpdate (String) calls).

When implementing these validations, we consider several syntaxes that should be validated: double and single quotes, SQL Injection, and XSS. In the HTTP requests, we validate the parameter value to avoid code injection and invalid HTML tags. For DB queries, the validation is made by analyzing the query string to prevent "always true" comparisons and comments.

When validating the HTTP requests, we prevent SQL Injection by removing any single or double quotes sent by the user. As a result, using the same example as before, for the user validation:

```
SELECT * FROM users WHERE user='alice' and password = 'toto' or 'a' = 'a'
```

The attacker should have input `alice` as the user and `toto' or 'a' = 'a` as the password. AProSec would validate this and change the password to `toto\' or \'a\' = \'a` taking the whole string as the password and not as two operations.

```
SELECT * FROM users WHERE user='alice' and password = 'toto\' or \'a\' = \'a'
```

As for the XSS, all the tags the user may input are transformed to HTML code preventing the attacker from introducing any tags. If the administrator wants to allow an HTML tag from the user, these tags are transformed to *safe tags* (explained in Section 4.2). Using the XSS example, in the input:

```
<script>
  for (q=0; q < 1000; q++)
        window.open(http://www.hot.example);
</script>
```

The `<script>` tag would be transformed into `&lt;script&gt;` allowing the browser to print it as text an not interpret it as a script. This includes other validations explained in this section.

## 4.2 The SQL Analyzer

Once the input got through the filter, if it is going to be used for a database query, it will be validated again in the context of the query. This helps to prevent unsafe queries to the database in case any malicious input got through the previous filters.

In order to validate JDBC requests, AProSec checks that the queries don't contain any comments or "always true" comparisons by not allowing queries like:

```
'value' = 'value'
'value' != 'value2'
table1.field1 = table1.field1
login = 'admin' -- ' and password = ''
```

Doing this, the SQL Analyzer enforces the application's security by not allowing unwanted code to be sent to the database and executed.

## 4.3 Configuration of the AProSec aspect

Even though single and double quotes are part of the SQL injection, the AProSec aspect manages them separately. We define all the validations that can be done, but the administrators can decide which ones to use by using the configuration file.

The configuration file is in XML format and is described in Figure 2.

```
<?xml version="1.0"?>
<!DOCTYPE validator [
<!ELEMENT validator
(validateQuotes,validateApost,validateSQLInj,valid
ateXSS,validTag*)>
 <!ELEMENT validateQuotes (#PCDATA)>
 <!ELEMENT validateApost (#PCDATA)>
 <!ELEMENT validateBackslash (#PCDATA)>
 <!ELEMENT validateSQLInj (#PCDATA)>
 <!ELEMENT validateXSS (#PCDATA)>
 <!ELEMENT validTag (#PCDATA)>
]>
```

**Figure 2: The configuration file**

In order to define the validations to make, we define a set of `ELEMENT` with the following meaning:

`validator`: This is the root element.

`validateQuotes`: To validate double quotes (") from a parameter. If this option is enabled, every time the applications receives a form or URL parameter, it will convert the double quote (") to "backslash double quote" (\").

validateApost: To validate single quotes (') from a parameter. If this option is enabled, every time the application receives a form or URL parameter, it will convert the single quote (') to "backslash single quote" (\').

validateBackslash: To validate backslash (\) from a parameter. If this option is enabled, every time the application receives a form or URL parameter, it will convert the backslash (\) to "double backslash" (\\).

validateSQLInj: To activate the SQL Analyzer and validate database queries with certain rules. If this option is enabled, every time the application issues a database call, the query is validated to prevent unexpected queries to execute.

validateXSS: To validate user input for XSS attacks. If this option is enabled, every time the application receives a form or URL parameter, this parameter is validated and all the HTML tags are transformed into safe tags.

validTag: To accept certain HTML tags. If this option is enabled and the validateXSS option is enabled too, then for every tag found in the parameter, this validation checks if it should accept the tag and transform it to a safe tag. This tag must be used for every HTML tag the administrator wants to accept.

A *safe tag* is the one that will not be printed as an HTML tag. For example, if a parameter contains the tag "<a href='#'> LINK </a>", the filter will transform it into "<a href='#'> LINK</a>", allowing the tag to be safely displayed. To enable an option, the value "TRUE" (case insensitive) should be used as the tag value. Any other value will disable the option. If an element is not present, then the default values are taken. The default values are all TRUE, without accepting any HTML tags.

Valid tags cannot contain an on* family element; if it does, it will be removed. If we are accepting the <a> tag, the input: This is <a href="#" onClick="alert('Thank you!');">a link</a>. Will be transformed as: This is <a href="#">a link</a>. Also, no parameter value can contain the words "javascript", "vbscript" nor "tcl", to prevent attacks like <img src="javascript:alert('Oops!');">.

## 4.4 Weaving with AspectJ

AspectJ [3] is the most widely used language for aspect-oriented programming. It defines an extension of the Java programming language for dealing with aspects. The AspectJ compiler handles Java source code or bytecode, weaves them with aspects, and generates some bytecode that can then be executed with a standard Java virtual machine.

Our first approach is made using AspectJ as the AOP framework, Tomcat as the application server and MySQL as the database manager. The code for intercepting the calls in AspectJ is described in Figure 3.

```
pointcut dbWrite(String query): (call(*
java.sql.Statement.addBatch(String))
  || call(* java.sql.Statement.execute(String))
  || call(* java.sql.Statement.executeQuery
(String))
  || call(* java.sql.Statement.executeUpdate
(String)))
  && args(query);
pointcut getParameter(): call(String
javax.servlet.http.HttpServletRequest.getParameter
(String));
Object around(String query): dbWrite(query){
  Object ret = validator.Validator().validateQuery
(proceed());
  return ret;
}
String around(): getParameter(){
  return new validator.Validator().validate
(proceed());
}
```

**Figure 3: The intercepting code with AspectJ**

In AspectJ the aspect is defined using the extended java language in a *.aj* file. By using the new expressions of the language we declare our pointcuts specifying the calls to be intercepted. With our pointcuts defined, we then call the validator to verify that the parameter or query is not dangerous.

## 4.5 Weaving with JBoss AOP

JBoss AOP [4] is a framework for programming aspect-oriented applications in Java. It can be used as a standalone framework or embedded in the JBoss J2EE server. Web applications running on this server can then take advantage of the aspect-oriented features of the framework. JBoss AOP is an open-source project that can be downloaded from http://www.jboss.org/products/aop

By using JBoss AOP, a vulnerable application can now be protected at compile time or at runtime by applying the security aspects. Both modes were tested. The main advantage of the load time (or runtime) mode is that the application doesn't need any manipulation before getting it in the WAS. Using the compile time mode, we need to recompile the source files and then package them before getting them to run in the WAS.

The JBoss code for intercepting the calls is described in Figure 4.

```
<aop>
  <bind pointcut="call(java.lang.String
$instanceof{javax.servlet.http.HttpServletRequest}
->getParameter*(java.lang.String))">
    <interceptor
class="interceptors.HTTPInterceptor"/>
  </bind>
  <bind pointcut="call(*
$instanceof{java.sql.Statement}-> addBatch*
(java.lang.String))">
    <interceptor class=
"interceptors.QueryInterceptorQuery"/>
  </bind>
  <bind pointcut="call(* $instanceof
{java.sql.Statement}-> execute*
(java.lang.String))">
    <interceptor
class="interceptors.QueryInterceptor"/>
  </bind>
  <bind pointcut="call(* $instanceof
{java.sql.Statement}-> executeQuery*
(java.lang.String))">
    <interceptor class=
"interceptors.QueryInterceptor"/>
  </bind>
  <bind pointcut="call(* $instanceof
{java.sql.Statement}-> executeUpdate*
(java.lang.String))">
    <interceptor class=
"interceptors.QueryInterceptor" />
  </bind>
</aop>
```

**Figure 4: The intercepting code with JBoss AOP**

When using JBoss AOP we define our aspect using a XML file. Here we specify the call we want to intercept and the class we want to call when intercepted. This class will then call the validator to verify the parameters and queries.

## 4.6 Experimentation results

We developed a vulnerable online bookstore, to test the AProSec aspect. First we tried all sorts of SQL Injection and XSS attacks to see how the application behaved. Then we protected it with AProSec using two approaches: AspectJ and JBoss AOP. After using AProSec we attacked the application again, but were unable to bypass the application's security.

For example, let assume than an attacker tries to input the following query in order to obtain information as a system administrator:

```
  select * from users where login='admin' - - and 'pwd=' ';
```

The query will not be processed by the database because it contains a commentary inside it. The SQL analyzer will detect it and will refuse to pass it to the database manager.

In another example the attacker will try to obtain information using a query that contains a statement that is always true.

```
 select * from users where login='admin' and pwd='' or 1=1;
```

The analyzer will detect that there is a statement that always is true and will refuse to process it.

Both frameworks, AsoectJ and JBoss AOP, will help to reach our goal, but since we prefer to keep the aspect working without the need of the source code, the runtime weaving sounds as a better option. This way, even if we don't have access to the source code we can still improve our applications' security.

# 5. RELATED WORK

## Security approaches for SQL injection and XSS

The best way to be protected against SQL attacks is to inspect all the data the user introduces to the application. Most of the work in this area attempts to limit the way in which a pre-programmed query will be used, allowing only the sentence that the programmer wants to define.

In [6] the authors propose to use a parsing tree that represents the parsed SQL query. Once the user introduces the required data, a new parsing tree is generated and compared with the first one. An SQL injection attack will produce a different tree.

The AMNESIA project [7] defines a model for detection of illegal SQL queries, before they are executed by the DBMS. In the first phase, the source code is analyzed in order to generate the model that contains the valid SQL queries. In a second phase, a real time monitor compares the SQL generated by the program with those stored in the model.

SQL DOM technique, described in [8], is a set of classes that are strongly-typed to a database schema. Instead of string manipulation, these classes generate SQL statements. The solution is based on an executable called sqldomgen, which generates a dynamic link library (DLL) based on the structure of the database. The DDL contains classes that will be used to construct dynamic SQL statements without manipulating any strings.

In [9] the authors propose a randomization of the instruction set. They create an execution environment that is unique to the running process. In order to achieve this, the original opcodes of the computer server are transformed by a random key. If an attacker tries to inject code and it does not know the key, the machine will not execute this code, causing a runtime exception.

Another solution is the use of application IDS (Instruction Detection System). This kind of IDS is oriented to supervise specific applications, including SQL applications. The authors in [10] propose to use a Network IDS in order to look for invalid SQL statements in the network traffic.

The advantage of AProSec, in comparison with the other works, is that it is based on AOP and it considers both, SQL Injection and XSS in the same aspect. Also, when using JBoss AOP it provides runtime weaving, allowing the administrator to incorporate AProSec without recompiling the application. Once the application is running with AProSec, any change in the configuration file will be taken during runtime, without stopping the application at any moment.

## 5.1 AOP and Security

The domains of aspects and security have already been the subject of several works. Among the security related functionalities that have been the topic of an aspect-oriented development, one can find: access control [15] [16] [17], encryption [12] [14], the adding of digital signatures [13], authorization [14] and authentication [14]. Most of the implementations described in these studies, such as [13] [14] [16], rely on AspectJ.

The work presented at [18] is closest to the objectives of our project. The authors propose an aspect to detect cross-site scripting. Their approach relies on sanitizing, i.e. replacing special characters by quoted ones, the input data submitted by users to web applications. The authors take the case of servlet-based web applications. When data is submitted to a servlet, one of the issues which are raised consists in determining whether it comes from an end-user or whether it comes from another servlet which delegates the request by mean of the transfer mechanism provided by the servlet container. In the latter case, data is supposed to be trustworthy as it simply originates from another part of the application. In this case, the sanitizing can be skipped in order to save computation time. To achieve this, the authors propose to extend the syntax of the AspectJ pointcut language with a new construct to detect data flows: the servlet input is sanitized if and only if it is written back on the servlet output stream. As far as we know, this data flow operator remains at the level of a proposal and has not been implemented. Furthermore, it remains to be seen in what circumstances this solution is more efficient than a solution that would sanitize all input streams regardless of their origin.

# 6. CONCLUSION

We have presented our approach for writing a security aspect in a web application server. This aspect detects SQL injection and XSS attacks in requests. As an advantage to usual solutions, this aspect allows the interception of all database accesses and validates them with its SQL Analyzer before dangerous information is stored. Moreover, the AProSec aspect can be parameterized. The administrator doesn't need to recompile the code and can freely decide which validations to apply to each web application. We have described our two experimentations, one with AspectJ and another with JBoss AOP.

With our approach, an aspect allows a clear separation of the security code and the WAS code. The initial code of the WAS

was not modified. By this way the aspect will be able to evolve independently. We only have to program it once for all web applications.

For further study, a first approach would be to add path traversal attack detection. The *path traversal* of a file is an attack in which, through request, the user provides information concerning the access path of a file (e.g., "../../target_dir/target_file"). This kind of attack tries to access files that shouldn't be accessible. These attacks can be sent in the form of a URL or of an entry such that it can have access to a given file. Second, cryptography issues can be added to applications in order to protect the disclosure of data for unauthorized parts. AOP will also take care of the key encryption management, and the encryption/decryption processes. This will be transparent for the users and their e-mails will be safe. Authentication can be added to, in order to accept any kind of known applications, token, or biometric. Finally, we plan to design and develop a more expressive pointcut language for security by the definition of an Aspect Specific Language (ASL).

## 7. ACKNOWLEDGMENTS

This work is partially funded by the Franco-Mexican Laboratory on Informatics (LaFMI) (http://lafmi.imag.fr/).

## 8. REFERENCES

[1] OWASP Top Ten Most Critical Web Application Security Vulnerabilities, http://www.owasp.org

[2] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, J. Irwin. *Aspect-Oriented Programming*. Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP'97). LNCS 1241. pp 220-242. June 1997. Springer-Verlag.

[3] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. Griswold. Overview of AspectJ. Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP'01). LNCS 2072. pp 327-353. June 2001. Springer-Verlag.

[4] M. Fleury, F. Reverbel. *The JBoss Extensible Server*. Proceedings of the 4th ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'03). LNCS 2672. pp 344-373. June 2003. Springer-Verlag.

[5] J. Viega, J.T. Bloch and P. Chandri, Applying Aspect-Oriented Programming to Security, Cutter IT Journal, Volume 14, No. 2, pp. 31-39, 2001 10

[6] G. Buehrer, B. Weide, P. Sivilotti, Paolo, *Using Parse Tree Validation to Prevent SQL Injection Attacks*, Proceedings of the 5th international workshop on Software engineering and middleware SEM '05, p. 106 – 113, September 2005.

[7] W. Halfond, A. Orso, *AMNESIA: Analysis and Monitoring for Neutralizing SQL – Injection Attacks.* In Proceedings of 20th ACM International Conference on Automated Software Engineering (ASE), Nov. 2005. 7, 2005, p. 174 – 183.

[8] R. McClure, I. Krüger, *Sql Dom: Compile Time Checking of Dynamic SQL Statements*. Proceedings of the 27th international conference on Software engineering. p. 88 – 96, May 2005.

[9] Kc, Gaurav, A. Keromytis, V. Prevelakis, *Countering Code-Injection Attacks With Instruction-Set Randomization*. CCS'03, Proceedings of the 10th ACM conference on Computer and communications security, p.272 – 280, October 2003.

[10] K. Mookhey, N. Burghate, *Detection of SQL Injection and Cross-site Scripting Attacks*. SecurityFocus. Marzo 17, 2004.

[11] Workshop for Application-level Security (AOSDSEC) @ the 3rd International Conference on Aspect-Oriented Software Development (AOSD'04). March 2004. Lancaster, UK.

[12] G. Bostrom. *Database Encryption as an Aspect*. In 11.

[13] R. Laney, J. van der Linden, P. Thomas. *Evolution of Aspects for Legacy System Security Concerns*. In 11.

[14] M. Huang, C. Wang, L. Zhang. *Toward a Reusable and Generic Security Aspect Library*. In 11.

[15] T. Verhanneman, F. Piessens, B. De Win, W. Joosen. *View Connectors for the Integration of Domain Specific Access Control*. In 11.

[16] B. De Win, F. Sanen, E. Truyen, W. Joosen, M. Südholt. *Study of the Security Concern*. Network of Excellence on Aspect-Oriented Software Development. Milestone 9.1. July 2005.

[17] B. De Win, W. Joosen, F. Piessens. *AOSD & Security: A Practical Assessment*. Workshop on Software Engineering Properties of Languages for Aspect Technologies (SPLAT) @ AOSD'03. pp 1-6. Boston, USA. March 2003.

[18] K. Kawauchi, H. Masuhara. *Dataflow Pointcut for Integrity Concerns*. In 11.