Using Aspect Programming to Secure Web Applications

Gabriel Hermosillo - Roberto Gomez
ITESM-CEM/Dpto. Ciencias Computacionales, Edo. de Mexico, Mexico
Email: {ghermosillo, rogomez}@itesm.mx

Lionel Seinturier - Laurence Duchien
University of Lille- LIFL- INRIA Project ADAM, Villeneuve d'Ascq, France
{lionel.seinturier, laurence.duchien}@lifl.fr

***Abstract*—As the Internet users increase, the need to protect web servers from malicious users has become a priority in many organizations and companies. 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 experimented this aspect with AspectJ language and JBoss AOP. By 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.**

***Index Terms*—Aspect-oriented programming, security, SQL injection, cross site scripting, design of web applications, reuse of aspect, dynamic weaving.**

I. Introduction

In the last years, the Internet web servers activity has increased. 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 on web applications, highlights the weakness of these software systems. OWASP's Top Ten listing references two common attacks on web applications: 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. This 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 personal or secret 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 consideration 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 concerns separation in software systems and for adding crosscutting functionalities without changing the business logic of the software. AOP provides specific language mechanisms that make it 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 Cross Site Scripting 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. These elements will appear as AspectJ aspects woven at compile-time and, in a second version, at run-time with the JBoss AOP framework [3] [4].

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 runtime.

The rest of this paper is organized as follows. Section 2 presents the motivation and principles of SQL Injection, XSS Cross Site Scripting and AOP. Section 3 provides the web application architecture. Section 4 defines our AProsec Aspect and its integration with the web server architecture. Section 5 details the difference between two weaving approaches with AspectJ and JBoss AOP. Section 6 shows the experimentation results. Section 7 describes some related work. Finally, section 8 concludes and discusses some future work.

II. Motivation and Principles

There are a lot of devises to implement perimetral security, firewalls, IDS, IPS, etc. Nevertheless a great number of organizations offer a web page to the public, leaving the port 80 open to any person who wants to access. The web service offers different kinds of services as information about the company, but it is also a way to send data to the organization. In most of the cases the system is developed "in house" with no security concerns at all.

In order to prevent insecurity in the design and the implementation of web applications, the OWASP (Open Web Application Security Project) created a list of the top 10 vulnerabilities, which represents the most critical web application security flaws. The primary aim of the OWASP Top 10 is to educate developers, designers, architects and organizations about the consequences of the most common web application security vulnerabilities.

The list includes cross site scripting, injection flaws, malicious file execution, cross site request forgery (CSRF), insecure direct object reference, information leakage and improper error handling, broken authentication and session management, insecure cryptographic storage, insecure communications, failure to restrict URL access.

This paper presents a scheme to prevent the two first vulnerabilities of the list, based on Aspect Oriented Programming. The main feature of our scheme is the fact that the initial code of the application does not need to be modified.

A. SQL Injection and XSS

SQL injection: The OWASP project explains that a SQL injection attack consists in finding a parameter that a web application sends to a database [1]. 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 login, password 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, when using the following SQL statement:

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

An attacker could transform by this 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.

Cross Site Scripting: The cross site scripting (XSS) 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. An attacker can also mislead the user to another website and try to extract personal or confidential information.

B. Aspect Oriented Programming

The domain of aspect-oriented programming (AOP) appeared in 1996 [1] [2]. 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.

An analogy of how the different concerns can be separated from the requirements using AOP, is how a prism separates a light beam into a spectrum of colors [5].

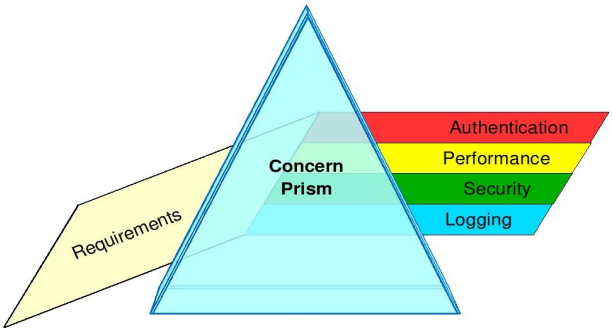

Figure 1: Prism analogy for concern separation

Fig. 1 shows the requirements as a light beam. When the beam enters the prism, all the concerns are separated so they can be developed and included independently. Authentication, performance, security and logging are commonly used concerns and are always spread through the application's code.

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.

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 [6].

III. Web Application Architecture

A web application architecture usually consists in three parts: a server where the web application is running, a database server where the application's information is stored and a bunch of clients willing to use the web application.

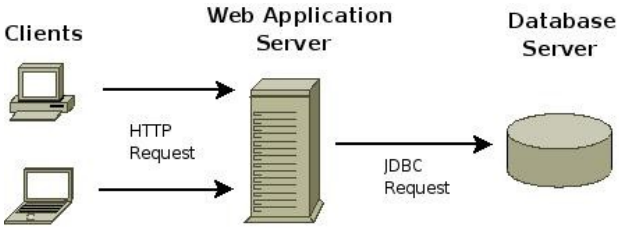

Figure 2: The architecture of an unprotected web application server

Fig. 2 shows how a web application server (WAS) interacts with clients and database servers. When the client sends a request, it goes directly to the WAS and if it isn't validated, it can cause some unexpected behavior. When needed, the WAS forms a new request using the unvalidated parameters from the client, forwarding the attack to the database server. This is how most of the attacks are done, using unvalidated entries.

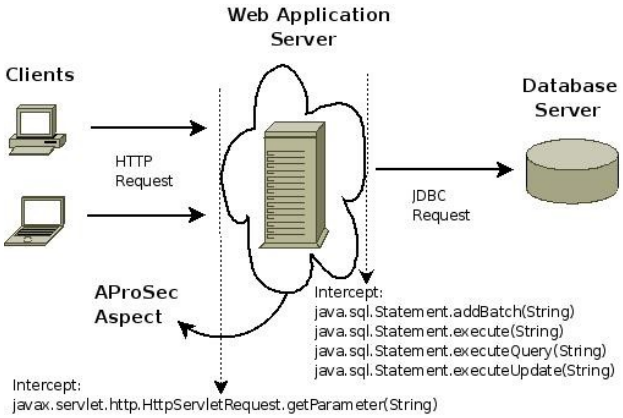

Figure 3: The architecture of a web application server protected by AProSec

Fig. 3 shows how AProSec protects the application by intercepting and validating all the requests from the client to the WAS and from the WAS to the database server. This prevents that any request goes unvalidated and that the attacks can get through. As shown in Fig. 3, AProSec surrounds the application, without having to change the application's source code.

IV. 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 IV-B. Finally, the pointcut part (where the code is added) allows the weaving with the web application. How this weaving is made will be described in Section V-A and V-B for each implementation.

A. 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 considered 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, semicolons 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. 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. By default, all the HTML tags are transformed into *safe tags*, but the administrator can configure the aspect to accept certain tags using the XML configuration file. In this case, the selected tag will remain unchanged, but still certain validations are done. For example, all the Javascript parameters that a tag may contain (like `onClick`, `onMouseOver`, etc.), would be removed.

When validating the JDBC requests, AProSec checks the queries so they do not contain any comments and prevents "always true" comparisons by not allowing queries like:

```
'value' = 'value'
'value' != 'value2'
table1.field1 = table1.field1
```

This helps to prevent any SQL Injection that bypasses the single and double quotes filter. In order to do this, AProSec intercepts the request sent to the database and analyzes all the conditions in the query, changing the "always true" conditions for "always false" ones. It also removes any semicolons (;) found in the query, to prevent the insertion of additional queries. Finally, AProSec detects comments in the query, preventing an attacker to comment any validations made.

B. Configuration of the AProSec aspect

This section describes the configuration file. 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.

It must be noticed that the whole JDBC validation is controlled by the *validateSQLInj* tag, so if it is enabled, the comments, semicolons and "always true" conditions are always checked.

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

Figure 4: The configuration file

Fig. 4 shows the XML configuration file in which 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” (\').

validateSQLInj: To validate the query for possible SQL Injection. If this option is enabled, every time the application issues a database call, the query is validated to prevent unexpected queries to execute, avoiding “always true” conditions and the use of semicolons and comments in the query.

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*, preventing the user input to be displayed in a dangerous war.

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. Even when this option is enabled, the parameters in the tag cannot contain any Javascript calls, as explained later.

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 “`&lt;a href='#'&gt; LINK&lt;/a&gt`”, 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 (like `onClick`, `onChange`, etc.); if it does, it will be removed. For example, 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!');">
```

If these scripts are found, they will also be removed, and the example would be transformed to:

```
<img src="">
```

V. Weaving the Aspect

A. Weaving with AspectJ

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

Our first approach is made using precisely AspectJ as the AOP framework, Tomcat as the application server and MySQL as the database manager.

Fig. 5 describes the calls in AspectJ. Here 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.

```
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.getParamet
er(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 5: The intercepting code with AspectJ

B. Weaving with JBoss AOP

JBoss AOP 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 [4]. 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 does not need any manipulation before getting it in the application server. Using the compile time mode, we need to recompile the source files and then package them before getting them to run in the application server.

Fig. 6 describes the JBoss code for intercepting the calls. 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.

```
<aop>
  <bind pointcut="call(java.lang.String
$instanceof{javax.servlet.http.HttpServletReques
t}->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 6: The intercepting code with JBoss AOP

VI. Experimentation Results

We developed a vulnerable online bookstore, to test the AProSec aspect. It is a simple application with a poor login, a catalog with a search engine and a message board. The login and the catalog are ideal to exploit the SQL Injection vulnerabilities. Using the login we could attack the restrictions of having a valid account, while the search engine, combined with the catalog can be exploited with “Union” attacks to display restricted info to the attacker. The message board is easy to attack with XSS, by inserting HTML tags in the comments that cause unexpected behavior in the user's browser.

First we tried all sorts of SQL Injection and XSS attacks we had information about to see how the application behaved. Then we protected it with AProSec using two approaches: AspectJ and JBoss AOP. After using AProSec we tried the same attacks and even some more, and were unable to bypass the application's security.

Our test database consists in two tables: users (login, password, admin_flag) and products (name, description, price). When using the product search in the application, the search would create the following query:

```
select * from products where name like
'%parameter%';
```

When the application runs without the aspect, an attacker could use the catalog search engine to display the users and passwords of the application.

Figure 7: SQL Injection using the search field

Fig. 7 shows the union attack that will return all the products and the users from the application. This input would create this query:

```
select * from products where name like '%' union
select * from users#;
```

This query would make the application show the users login and password.

Name	Description	Price
one book	a book from product	$40.00
a magazine	just another product	$10.00
videogame	for the fun of it!	$50.00
user1	pwd1	$0.00
user2	pwd2	$0.00
admin	superpwd	$1.00

Figure 8: Display of products and users

If the application is running with AProSec, it will not show any informations from the users table and the query would be transformed to:

```
select * from products where name like '%\'
union select * from users#;
```

Both frameworks (AspectJ and JBoss) 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, compared to the compile time approach. This way, even if we do not have access to the source code we can still improve our applications' security.

Typically, the downside of inserting additional validations to an existing application is that the performance can be badly damaged, but by using good design patterns during the development of AProSec, this was not an issue.

We recorded the time that database transactions took to respond with and without AProSec, and the average difference between both was of about 0.0035 seconds, using all the possible validations in the aspect. These tests were done on a database (MySQL) with over 100,000 records and using a simple workstation for both, Database and Web Application Server. We expect that performance will be even less diminished when using a dedicated server.

VII. Related Work

A. *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 other project, the authors propose to use a parse tree that represents the parsed SQL query [7]. In order to achieve this, the SQL grammar has to be known. This technique produces one parse tree with the original query, including the expected user input. Once the user introduces the required data, a new parse tree is generated and compared with the first one. Since an SQL injection attack will produce a different tree, the comparison will show the differences and detect the attack.

AMNESIA project is a tool that detects and prevents SQL injection attacks by combining static analysis and runtime monitoring [8]. It 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, if they do not match the model, the queries are prevented from executing on the database. It's success depends on the accuracy of the model generated during the first phase, and can be degraded when using certain types of code obfuscation or query development techniques [9].

SQL DOM technique is a set of classes that are strongly-typed to a database schema [10]. 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. This technique requires the learning and use of a new programming paradigms or query-development process and do not provide any protection for legacy systems [9].

A randomization of the instruction set is proposed by another team [11]. 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. The security of this technique is dependent on the attacker not being able to discover the key. It requires the developers to modify the application to use the randomized instructions.

Another solution is the use of application IDS (Instruction Detection System) [12]. This kind of IDS is oriented to supervise specific applications, including SQL applications. The authors propose to use a Network IDS in order to look for invalid SQL statements in the network traffic. With these type of systems the SQL commands that will be executed can be deciphered, and depending on which columns and tables are been accessed, the IDS can conclude if it is an attack or not [13]. This techniques can provide no guarantees about their detection abilities because their success is dependent on the quality of the training set used [9].

A Context-Sensitive String Evaluation (CSSE) project presents a detection tool that does not require the application source code to be modified [14]. It modifies the PHP runtime environment to detect injection attacks on all the applications running on the server. By modifying the runtime environment, the portability gets affected [9]. The authors accept that a separate implementation is needed for every platform.

The advantage of AProSec, in comparison with the other projects, 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.

B. 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, encryption, the adding of digital signatures, authorization and authentication [15] [16] [17] [18] [19] [20]. Most of the implementations described in these studies rely on AspectJ [16] [17] [19].

A UML aspect-oriented profile is proposed by Jian Zhu and Zulkernine to model attack scenarios [21]. The model uses class diagrams and state machine diagrams to represent the static attributes and the dynamic behavior of the intrusions and the process of detecting these intrusions. Their framework consists of five stages: identify vulnerabilities and attacks; model attack scenarios using UML; generate the intrusion detection aspect (IDA) code using an aspect code generator; weave aspects into the target system; test and deploy the integrated system. The aspect-oriented attack scenario model is exported to a XML Metadata Interchange (XMI) file, which is used as input by the code generator to create partial code for the IDA.

The work of Kawachi and Masuhara is the most related to our objectives [22]. 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.

VIII. 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 to a web application and from this to a database. It allows the interception of all database accesses and the validation of them before dangerous information is stored. Moreover,

the AProSec aspect can be parameterized. The administrator does not 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 Language and another with JBoss AOP.

With our approach, an aspect allows a clear separation of the security code and the web application code. The initial code of the web application was not modified. By doing this, 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).

Acknowledgments

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

References

[1] OWASP Top Ten Most Critical Web Application Security Vulnerabilities, http://www.owasp.org. Accessed in November 2007.

[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] R. Laddad. *I want my AOP. Separate software concerns with aspect-oriented programming*. JavaWorld. January 2002. http://www.javaworld.com/javaworld/jw-01-2002/jw-0118-aspect.html. Accessed in November 2007.

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

[7] 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.

[8] 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.

[9] W. G. Halfond, J. Viegas, and A. Orso. *A Classification of SQL-Injection Attacks and Countermeasures*. Proceedings of the International Symposium on Secure Software Engineering (ISSSE 2006), March 2006.

[10] 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.

[11] 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.

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

[13] A. Newman. *App IDS guards databases*. Network World. October 2005. http://www.networkworld.com/news/tech/2005/102405techupdate.html. Accessed in November 2007.

[14] T. Pietraszek and C. V. Berghe. *Defending Against Injection Attacks through Context-Sensitive String Evaluation*. In Proceedings of Recent Advances in Intrusion Detection (RAID2005), September 2005.

[15] G. Bostrom. *Database Encryption as an Aspect*. Proceedings of AOSD 2004 Workshop on AOSD Technology for Application-level Security (AOSDSEC), March 2004.

[16] R. Laney, J. van der Linden, P. Thomas. *Evolution of Aspects for Legacy System Security Concerns*. Proceedings of AOSD 2004 Workshop on AOSD Technology for Application-level Security (AOSDSEC), March 2004.

[17] M. Huang, C. Wang, L. Zhang. *Toward a Reusable and Generic Security Aspect Library*. Proceedings of AOSD 2004 Workshop on AOSD Technology for Application-

level Security (AOSDSEC), March 2004.
[18] T. Verhanneman, F. Piessens, B. De Win, W. Joosen. *View Connectors for the Integration of Domain Specific Access Control*. Proceedings of AOSD 2004 Workshop on AOSD Technology for Application-level Security (AOSDSEC), March 2004.
[19] 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.
[20] 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.
[21] Z. Zhi Jian, M. Zulkernine. *Towards an Aspect-Oriented Intrusion Detection Framework*. Computer Software and Applications Conference (COMPSAC 2007), July 2007.
[22] K. Kawauchi, H. Masuhara. *Dataflow Pointcut for Integrity Concerns*. Proceedings of AOSD 2004 Workshop on AOSD Technology for Application-level Security (AOSDSEC), March 2004.

Lionel Seinturier is professor of computer science at the University of Lille since September 2006. Before that, he was associate professor of computer science at the Univesity of Paris 6. He got his PhD at CNAM, Paris in 1997 and his engineer diploma, both in computer science, at IIE, Evry, France in 1993. His research activities deal with middleware design and implementation. He is one of the three co-authors of the book Foundations on AOP for J2EE Development (APress, 2005) and of more than 30 publications in international conferences and journals.

Laurence Duchien is currently full professor at the department of computer science at University of Lille, France since 2001 and she is the head of the INRIA team-project ADAM (Adaptive Distributed Applications and Middleware) http://adam.lifl.fr. Before, she was associate professor of computer science at CNAM, Paris. She got her PhD at University of Paris 6 in 1988. Her research interests are centered on the area of component-based architecture design, software evolution and model driven engineering. Her research interests include formal description and development techniques for component-based and service-oriented architecture modeling, analysis, transformation, and evolution for distributed applications. She currently involves in ERCIM Group Software Evolution and in AOSD-Europe NoE.

Roberto Gomez is proffesor of computer science at the ITESM-CEM in Mexico City. His research interests include cryptography, applications security. He received a PhD in computer science from the Univeristy of Paris 8 in 1995. During his PhD studies all his research work took place at INRIA Rocquencourt laboratories. Roberto is a member of the ACM, IEEE and IARC (International Association for Cryptologic Research).

Gabriel Hermosillo is a Master candidate at the ITESM-CEM in Mexico City. He is a certified Java programmer and works as a full-time system developer and analyst at the Campus. His research interests include security applied on web applications and software development.