Global Journals LaTeX JournalKaleidoscope™
Artificial Intelligence formulated this projection for compatibility purposes from the original article published at Global Journals. However, this technology is currently in beta. *Therefore, kindly ignore odd layouts, missed formulae, text, tables, or figures.*

Towards full protection of web applications based on Aspect Oriented Programming

Dr. Elinda Kajo Mece[1] and Lorena Kodra[2]

[1] University of Tirana

Received: 9 April 2012 Accepted: 30 April 2012 Published: 15 May 2012

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 a technique for detecting and preventing common attacks in web applications like Cross Site Scripting (XSS) and SQL Injection using an aspect oriented approach by analyzing and validating 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. We present an experimental evaluation by applying it to an insecure web application. The results of our tests show that our technique was able to detect all the attempted attacks without generating any false positives.

Index terms— symbolic information, artificial intelligence, Flow control, Architecture.

1 INTRODUCTION

ser and critically important company information is managed using web applications. For this reason, web applications serve as 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.

Two common attacks on this type of systems are Cross Site Scripting (XSS) and SQL Injection. SQL Injection is a technique where an intruder 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 intruder injects and executes scripts written in languages like JavaScript or VBScript. 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 makes it a good candidate for applying security to a system.

2 Author

? ? : Department of Computer Engineering, Polytechnic University of Tirana, Tirana, Albania. E-mails : ekajo@fti.edu.al, lorena.kodra@gmail.com

In this paper, we propose 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.

The rest of the paper is organized as follows. Section 2 presents principles of SQL Injection, XSS and AOP. Section 3 presents related work in this area and our proposed solution. Section 4 describes in detail 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.

3 II.

BACKGROUND a) SQL Injection SQL Injection consists in inserting malicious SQL commands into a parameter that a web application sends to a database in order to execute a malicious query. As a result, database contents can be corrupted or destroyed. The most popular techniques used in SQL injection are tautology, union, and comments.

The general idea behind tautology is finding a disjunction in the WHERE clause of a SELECT or UPDATE statement and inserting malicious code into one or more conditional statements so that they always evaluate as true. Let us 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' SELECT productName FROM products WHERE productID = '5' An attacker can use the UNION clause to modify the structure of this query to: SELECT productName FROM products WHERE productID = '5' UNION SELECT username, password FROM users As a result, 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 cut an SQL query and change its structure. 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. For example the following SQL statement:

SELECT * FROM users WHERE username = 'alice' and password = 'alice123' 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. b) Cross Site Scripting Cross Site Scripting (XSS) is an attack done towards the user's browser in order to attack the local machine, steal user information or to spoof the user identity. 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.

4 c) Aspect Oriented Programming and Security

5 Aspect

Oriented Programming is a programming paradigm whose aim is 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 means that when developers want to fix a bug 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.

6 III.

7 RELATED WORK AND PROPOSED SOLUTION

During recent years, different solutions have been proposed to address security issues in web applications. The most efficient way to protect against XSS and SQL Injection attacks is to inspect all the data the user inserts into the system, hence most of the work in this area treats user input.

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. We propose a system that performs a two-step validation of user input. In the first step it is validated syntactically 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 in the context of a query to check whether it contains always true statements, comments or combinations of SQL keywords. In contrast to the systems described above, our system analyzes directly user input before it is being used as part of an SQL query. This facilitates the analyzing process. 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. Our system does not generate false positives because it considers as attack the presence of a combination of SQL keywords and not the presence of a single SQL keyword such as "Union" that might be part of a legitimate user name.

8 SYSTEM ARCHITECTURE

Our system consists of three parts. The first and the most important part is an aspect called WebAppInputFilter that contains the logic of the whole defense process. It defines the advices that control the validation process as well as the steps 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. The second part consists of a validators class that validate against XSS and SQL Injection attacks the input defined in the advices. The third part consists of an encoder which encodes dangerous characters by converting them to their decimal equivalent, leaving them harmless.

The basic idea behind our technique is to capture user input and validate it by comparing it to predefined patterns. In the case of SQL Injection, in contrast with current solutions [1, 2, 3], the user input is validated before being used as part of a query. The final query is a combination of user input and a partial SQL statement defined by the developer. We consider as safe the part of the query that is defined by the developer, so there is no need to validate it and we only validate the user input part. This facilitates and speeds up the evaluation process.

The validation process happens in two steps. First the user input is validated to check whether it contains dangerous characters such as '<',' >', '=' and' -' that can be used to perform XSS and SQL Injection attacks. In the second step, the SQL Validator analyzes the input 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 shows the flow of information within the defense system. The aspect captures the user input string and sends it to the first analyzer. If the string is not dangerous it is passed on to the second validation step. If the string is dangerous it is send to the encoder. It encodes the dangerous characters and the result is passed to the SQL 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. This aspect is implemented in AspectJ [7]. This is the most widely used language for aspect oriented programming. It represents the extension of Java for dealing with aspects. The aspect defines pointcuts in the vulnerable points of the web application. It monitors the traffic in servlets and captures some specific calls that implement the The syntactic validator, analyzes separately each character of the user input string and acts as a filter that allows only characters 'a-z', 'A-Z', numbers '0-9', spaces and characters like "." and ",". The rest of the characters are considered dangerous and will be sent to the encoder.

The SQL Validator consists of several validation strings in the form of regular expressions that are matched against user input according to different possibilities of injecting SQL code into the user input field of the web application. The validation criteria include: always true comparisons (both string and numeric), presence of quotes or comments, keywords for executing stored procedures, combinations of SQL keywords like UNION, SELECT, DROP, INSERT, ALL, etc. As regards this least evaluation criterion, it protects in cases where no comments or always true statements are present in the query but it still may contain dangerous keywords that can execute arbitrary January 2012 operations in the database. We would also like to emphasize that the SQL Validator doesn't simply detect the presence of SQL keywords, but the presence of combined SQL keywords that would potentially modify the original structure of the query. This means that input strings that simply contain SQL keywords (like UNION) will not be considered dangerous unless they contain some other SQL keyword that would create a risk for SQL Injection. This eliminates the false positive case of detection when a legitimate user has for example the word "Union" in their name.

V.

9 EVALUATION RESULTS

We evaluated our system by using it against a vulnerable web application **??**8]. First we tried all sorts of SQL Injection and XSS injection attacks to see how the system behaved. Then we protected it using our system but were unable to bypass the application's security.

For example, let's assume that an attacker tries to input the following script into the web application: <script>alert(document.cookie)</script>

The system will detect the dangerous characters "<", ">", "(", ")" and "/" and encode them. In this way this input string will be considered as a simple string and not as a script and will not be interpreted by the browser. A wiser attack would be to encode the input string by using some encoding scheme (decimal, hexadecimal, octal,

Unicode, etc) prior to inserting it into the web application. For example, the above string in hexadecimal format (\xNN) would be:\x3c\x73\x63\x72\x69\x70\x74\x3e\x61\x6c \x65\x72\x74\x28\x64\x6f\x63\x75\x6d\x65 \x6e\x74\x2e\x63\x6f\x6f\x6b\x69\x65\x29 \x3c\x2f\x73\x63\x72\x69\x70\x74\x3e

Even in this case the attack wouldn't be successful because the system detects the usage of "\" and encodes the string to make it harmless. We tested our defense system by using other encodings (decimal, octal and Unicode) and none of the attacks were successful.

In the case of SQL Injection, let's assume that an attacker tries to inject a query that contains a statement that is always true into the system: SELECT * FROM user_data WHERE last_name = 'Smith' OR '1'='1'

The SQL Validator will detect that there is a statement that is always true and will delete this string without passing it to the web application.

In order to evaluate the impact of the defense system in the performance of the web application we measured its response time using [9] under two scenarios. We measured the response time first in the absence of any defense and then in the presence of our defense system. We used a mix of input strings: harmless, XSS attack and SQL Injection attack strings. For every scenario we used 356 POST and 104 GET requests which make a total of 460 requests. We executed the series of requests 5 times and measured the average response time. Our defense system introduced an average overhead of 2.11%. We feel that this is an acceptable level of overhead for use in many production environments and it will not be noticeable by the user.

10 VI. CONCLUSIONS AND FUTURE WORK

We have presented our approach for building a 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 separation of security concerns and allows the security system to be evolved independently from the web application to adapt to new attacks.

As an advantage to similar solutions, besides checking for comments and always true statements, our SQL Validator also checks for the presence of a combination of SQL keywords in the input string. This can protect in cases where comments or always true statements are not present in the query but it still may contain dangerous keywords that can execute arbitrary operations in the database. Our system does not simply check for SQL keywords but for a combination of them. This is considered as an advantage in eliminating false positives like in the case of having for example the word "Union" as part of a legitimate user name. Furthermore, in contrast to usual solutions, when protecting against SQL Injection our system analyzes directly the user input before being used as part of a query. There is no need to analyze the whole query because the other parts of it are defined by the developer and are considered safe. This has the advantage of facilitating and speeding up the evaluation process.

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 [10]. [1] [2]

[1] January 2012© 2012 Global Journals Inc. (US)IV.

[2] © 2012 Global Journals Inc. (US) Global Journal of Computer Science and Technology Volume XII Issue I Version I

1

Figure 1: Fig. 1 :

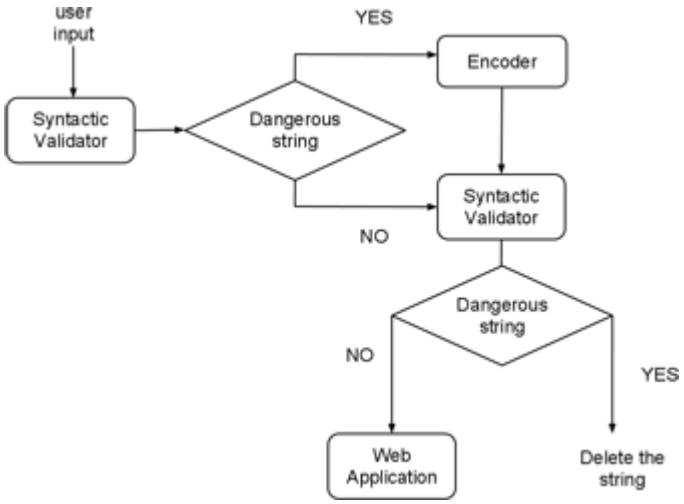

Figure 2:

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

[Jmeter] , Apache Jmeter . http://jakarta.apache.org/jmeter/JBosswww.jboss.org/jbossaop

[Zh et al. ()] 'A model-based aspect-oriented framework for building intrusionaware software systems'. J Zh , Z Zhi , Mohammad . *Information and Software Technology*, 2009. 51 p. .

[Gabriel et al. ()] 'AProSec: An aspect for programming secure web applications'. H Gabriel , G Roberto , S Lionel , D Laurence . *Proceedings of the The Second International Conference on Availability, Reliability and Security*, (the The Second International Conference on Availability, Reliability and Security) 2007. p. .

[Engin et al. ()] 'Client-side cross-site scripting protection'. K Engin , J Nenad , K Christopher , V Giovanni . *Computers & Security* 2009. 28 (7) p. .

[Etienne and Pavol ()] 'Preventing SQL Injections in Online Applications: Study, Recommendations and Java Solution Prototype Based on the SQL DOM'. J Etienne , Z Pavol . *OWASP AppSec Conference*, 2008.

[Dimitris and Spinellis ()] 'SDriver: Location-specific signatures prevent SQL injection attacks'. M Dimitris , Diomidis Spinellis . *Computers & Security* 2009. 28 p. .

[Matias et al. ()] 'Watch What You Write: Preventing Cross-Site Scripting by Observing Program Output'. M Matias , L Edward , W Jacob , C Brian . *OWASP AppSec Conference*, 2008.