Detection and Prevention of Web Application Security Attacks

Sandra Sarasan
Department of Computer Science & Engineering, University of Calicut, Calicut, India
E-mail : sandrasarasan002@gmail.com

Abstract – Today’s Web applications can contain dangerous security flaws. The global distribution of these applications makes them prone to attacks that uncover and maliciously exploit a variety of security vulnerabilities. Research reports indicate that more than 80 percent of the web applications are vulnerable to security threats. User friendly web applications are developed to increase the customer base and hackers utilize the features provided by the web applications to inject their malicious code. Web applications might contain security vulnerabilities that are not seen to the owner of the application. This paper presents multiple solutions to prevent web applications from the major security attacks such as SQL Injection and Cross Site Scripting. Each of the solutions have their own strengths and weaknesses, and the developers must choose the solutions according to their software development requirements.


I. INTRODUCTION

Security vulnerabilities in web applications have been a significant problem for the computer industry for decades. In recent years the development of Internet has huge impact on commerce and culture. As each coin has two sides; the World Wide Web has both pros and cons. On one hand where the Web can dramatically lower costs to organizations for distributing information, products, and services, on the other computers that make up the Web are vulnerable. Today almost everyone in touch of computer is connected online and to serve millions of users, huge amount of data is stored in databases connected to some web application all across the globe. These users keep on inserting, querying and updating data in these databases. For all these operations a well-designed user interface is very important. Furthermore, these applications play a vital role in maintaining security of data stored in these databases. Web applications which are not very secured allow well-designed injection to perform unwanted operations on back-end database. An unauthorized user may be able to exploit this situation by damaging or with theft of trusted users sensitive data stored here. The maximum damage is caused when an attacker gains full control over a database or web application as there is a chance of system being fully destroyed. The sensitive data on one’s system could be transferred to any third party by injecting a class of vulnerability in trusted sites dynamically generated pages. This third party could be an attacker’s server. This mechanism could also be used in avoiding cookie protection mechanism or same-origin-policy.

SQL (Structured Query Language) Injection and Cross-Site Scripting (XSS), the two top most attacks according to OWASP (Open Web Application Security Project), has been used frequently to implement the attacks. SQL Injection is a method of injecting SQL meta-characters or commands inside web-based input fields so that execution of the backend SQL queries can be manipulated. Web servers belonging to an organization is primary target of these kinds of attack. Cross site scripting (XSS) vulnerability occurs when a web application uses inputs received from users in web pages without properly checking them. This allows an attacker to inject malicious scripts in web pages via such inputs such that the scripts perform malicious actions when a client visits the exploited web pages. Such an attack may cause serious security violations such as account hijacking and cookie theft. Current approaches to mitigate this problem mainly focus on effective detection of XSS vulnerabilities in the programs or prevention of real time XSS attacks. As more sophisticated attack vectors are being discovered, vulnerabilities if not removed could be exploited anytime. XSS usually affects victim’s web browser on the client-side where as SQL injection occurs in server side. These vulnerabilities could be exploited by SQL injection or XSS to gain control over the online web
application database. In this paper we focus on various SQL injection and XSS attacks and approaches for their detection and prevention.

II. VULNERABILITY TYPES

Currently, there are many types of vulnerabilities that vary in terms of complexity, detection and recovery. This paper tackles with the following types of attacks.

A. SQL Injection

SQL Injection is a type of injection or attack in a Web application, in which the attacker provides SQL code to a user input box of a Web form to gain information access from databases. The attacker’s input is transmitted into an SQL query in such a way that it will form an SQL code. An unauthorized access to the data by a crafted user can threat in their confidentiality, integrity, and authority. As a result, the system could bear heavy loss in giving proper services to its users or it may face complete destruction. There are some classification of SQL injection types [1], [2] such as:-

1) Tautology: This attack bypasses the authentication and access data through vulnerable input field using "WHERE" clause by injecting SQL tokens into conditional query statements which always evaluates to true.

Example: SELECT * FROM <tablename> WHERE userId = <id> AND password = <wrongPassword> OR 1=1;

2) Union queries: The Union keyword in SQL can be used to get information about other tables in the database. And if used properly this can be exploited by attacker to get valuable data about a user from the database.

Example: SELECT * FROM <tablename> WHERE userId = <id> AND password = <rightPassword> UNION SELECT creditCardNumber FROM CreditCardTable;

3) Piggy-backed Queries: This is the kind of attack where an attacker appends ‘;’ and a query which can be executed on the database. It could be one of the very dangerous attacks on database which could damage or may completely destroy a table. If this attack is successful then there could be huge loss of data.

Example: SELECT * FROM <tablename> WHERE userId = <id> AND password = <rightPassword>; DROP TABLE <tablename>;

4) Blind Injection: It is difficult for an attacker to get information about a database when developers hide the error message coming from the database and send a user to a generic error displaying page. It’s at this point when an attacker can send a set of true/false questions to steal data.

Example: SELECT name FROM <tablename> WHERE id=<username> AND 1 = 0 – AND pass =

SELECT name FROM <tablename> WHERE id=<username> AND 1 = 1 – AND pass =

Both the queries will return an error message in case the web application is secure, however if there is no validation for input then the chances of injection exist. If attacker receives an error after submitting the first query, he might not know that, was it because of input validation or error in query formation. After that on submission of the second query which is always true if there is no error message then it clearly states that id field is vulnerable.

5) Timing Attacks: In this kind of attack timing delays are observed in response from a database which helps to gather information from a database. SQL engine is caused to execute a long running query or a time delay statement with the help of if-then statement which depends on the logic that has been injected. It is possible to determine whether injected statement was true or false depending on how much time page takes to load. The keyword WAITFOR along the branches can cause response delay for a given time in a database.

Example: DECLARE @s varchar(500) SELECT @s =db_nameO IF (ascii(substring(@s, 1, 1)) & (power(3,0))) > O WAITFOR delay '0:0:10'

In this example, if the database used has a name with the first bit of the first byte as 1, then the database get paused for ten seconds. So, when condition is true this code is injected to produce response delay in time.

B. Cross-Site Scripting (XSS)

Cross-Site Scripting attacks are those attacks against web applications in which an attacker gets control of the user’s browser in order to execute a malicious script (usually an HTML(HyperText Markup Language)/JavaScript code) within the context of trust of the web application’s site [3]. As a result, and if the embedded code is successfully executed, the attacker might then be able to access, passively or actively, to any sensitive browser resource associated to the web application (e.g., cookies, session IDs, etc.). Typical
input sources that attackers manipulate include HTML forms, cookies, URLs, and external files. Attackers often favor JavaScript, but other kinds of client-side scripts such as VBScript (Visual Basic Script) and Flash, which browsers can interpret, could cause XSS. Different types of XSS attacks are:

1) **Persistent/ Stored XSS**: The persistent XSS vulnerability is a more devastating variant of a cross-site scripting flaw. It occurs when the data provided by the attacker is saved by the server, and then permanently displayed as "normal" pages returned to the users in the course of regular browsing, without proper HTML escaping.

An example of this is, with online message boards where users are allowed to post HTML formatted messages for other users to read.

2) **Non-persistent/ Reflected XSS**: The non-persistent cross-site scripting vulnerability is the most common type. These attacks occur when the data provided by a web client, most commonly in HTTP query parameters or in HTML form submissions, is used immediately by server-side scripts to generate a page of results for that user, without properly sanitizing the request.

An example is, the non-persistent XSS vulnerabilities in Google, could allow malicious sites to attack Google users who visit them while logged in.

3) **DOM-based XSS**: DOM-based vulnerabilities occur in the content processing stages performed by the client, typically in client-side JavaScript. The name refers to the standard model for representing HTML or XML contents which is called the Document Object Model (DOM). JavaScript programs manipulate the state of a web page and populate it with dynamically-computed data primarily by acting upon the DOM.

A typical example is a piece of JavaScript accessing and extracting data from the URL via the location.* DOM, or receiving raw non-HTML data from the server via XMLHttpRequest, and then using this information to write dynamic HTML without proper escaping, entirely on client side.

### III. PREVENTIVE MEASURES AGAINST ATTACKS

#### A. SQL Injection Defenses

Preventing SQL Injection requires keeping untrusted data separate from commands and queries. The preferred option is to use a safe API (Application Program Interface) which avoids the use of the interpreter entirely or provides parameterized interface. Beware of APIs, such as stored procedures that appear parameterized, but may still allow injection. If a parameterized API is not available, you should carefully escape special characters using the specific escape syntax for that interpreter. Positive or white list input validation with appropriate canonicalization also helps protect against injection, but is not a complete defense as many applications require special characters in their...
input. Thus SQL Injection defenses are broadly classified into two types:

- Defensive coding
- SQL Injection vulnerability detection

1) Defensive Coding:

The use of proper coding practices is a straightforward solution for SQL Injection vulnerability since it is due to insecure coding of web developers [4]. Some of the defensive coding methods are the following:

- Parameterized queries or stored procedures:

  Proper coding style of parameterized queries or stored procedures enforces developers to define the structure of SQL code first before including parameters to the query. Because parameters are bound to the defined SQL structure, no injection of additional SQL code is possible.

- Data type validation:

  Validating whether an input is string or numeric could easily reject type-mismatched inputs. This could also simplify the escaping process because validated numeric inputs need no further cleansing action and could be safely used in queries.

- Whitelist filtering:

  Developers often use blacklist filtering to reject known bad special characters such as "'" and ";" from the parameters to avoid SQL injection. However, the safer and recommended filtering approach is to accept only inputs that are known to be legitimate. This approach is suitable for well-structured data such as (email) addresses, dates, zip codes, social security numbers, etc. Developers could keep a list of legitimate data patterns and accept only the input data which match them.

Although these methods can defeat SQL injection, their applications are manual, labor intensive, error-prone, and difficult to be rigorous and complete. To solve these problems, one approach is to use SQL DOM(Document Object Model) - a set of classes that provide automated data type validation and escaping. Developers are required to provide SQL DOM with database schema and construct SQL statements using its APIs. This framework is useful when a developer needs to use dynamic query construction method, instead of parameterized queries for greater flexibility. Both defensive coding and SQL DOM require high user involvement. They are used in development stage, require developer training and code modification are manual. These two approaches have no vulnerability locating and verification assistance.

2) SQL Injection Vulnerability Detection:

- Code based vulnerability testing:

  It uses static analysis to track user inputs to database access points and generate unit test cases for these points [5]. Test cases contain SQL Injection attack patterns. One approach is to generate mutated queries using mutation operators to replace the original queries in the Web programs. Only test cases that contain adequate SQL injection attacks could kill these mutants. A tool developed automatically injects SQL Injection vulnerabilities into the web programs and generates SQL Injection attacks. These two approaches are fault-based techniques and are able to assess the effectiveness of the security mechanisms implemented in the applications under test based on the injected vulnerabilities/mutants that are breached. In general, this type of approaches aims to generate adequate test suite for testing SQL Injection vulnerabilities. However, they do not explicitly find vulnerable program points and manual inspection is required to locate them.

- Concrete attack generation:

  This type of approaches uses state-of-the-art symbolic execution techniques for automatic generation of test inputs that actually expose SQL Injection vulnerabilities in the programs. Symbolic execution is a well known technique for automatic generation of test inputs that could exercise various program points. It generates test inputs by solving the constraints imposed on the inputs along the path to be exercised.

  Traditionally, symbolic execution based approaches use constraint solvers which only handle numeric operations. Because inputs to Web applications are, by default, strings, if a constraint solver is able to solve a number of string operations applied to inputs, symbolic execution could be used to both detect the vulnerability of SQL statements which use inputs and generate concrete inputs that attack them. Another approach provides a powerful hybrid constraint solver for numeric and string inputs. The solver is used to solve path conditions that lead to SQL statements, and extract test inputs containing SQL Injection attacks from the solution pool. If one such test input is generated, the corresponding SQL statement is vulnerable.

B. XSS Defenses

Preventing XSS requires keeping un-trusted data separate from active browser content. The preferred
option is to properly escape all untrusted data based on the HTML context (body, attribute, JavaScript, CSS, or URL) that the data will be placed into. Developers need to include this escaping in their applications unless their User Interface framework does this for them. Positive or white list input validation with appropriate canonicalization (decoding) also helps protect against XSS, but is not a complete defense as many applications require special characters in their input. Such validation should, as much as possible, decode any encoded input, and then validate the length, characters, format, and any business rules on that data before accepting the input. XSS defenses can be broadly classified into three types:

- Defensive coding practices
- XSS testing
- Vulnerability detection

1) Defensive Coding:

Defensive coding validates and sanitize the user inputs to eliminate the XSS vulnerabilities that may arise due to improper handling of inputs [6]. Input validation ensures that user inputs conform to a required input format. There are four basic input sanitization options. Replacement and removal methods search for known bad characters (blacklist comparison). The replacement method replaces them with non-malicious characters, whereas the removal method simply removes them. Escaping methods search for characters that have special meanings for client-side interpreters and remove those meanings. Restriction techniques limit inputs to known good inputs (whitelist comparison). Checking blacklisted characters in the inputs is more scalable, but blacklist comparisons often fail as it is difficult to anticipate every attack signature variant. Whitelist comparisons are considered more secure, but they can result in the rejection of many unlisted valid inputs. OWASP has issued rules that define proper escaping schemes for inputs referenced in different HTML output locations. Defensive coding practices have some limitations such as: labor-intensiveness, prone to human error, and difficult to enforce in deployed applications.

2) XSS Testing:

Input validation testing could uncover XSS vulnerabilities in web applications. Specification-based input validation testing methods generate test cases with the aim of exercising various combinations of valid/invalid input conditions stated in specifications [7]. The sole dependency on specifications can be avoided by inferring valid input conditions through analysing input fields and their surrounding texts in client-side scripts. Code-based input validation testing methods apply static analysis to extract valid/invalid input conditions from server-side scripts. In general, the effectiveness of both specification- and code-based approaches relies largely on the completeness of specifications or the adequacy of generated test suites for discovering XSS vulnerabilities in source code. Only test cases containing adequate XSS attack vectors can induce original and mutated programs to behave differently. A fault-based XSS testing tool creates mutated programs by changing sensitive program statements, or sinks, with mutation operators. This fault-based approach generates adequate test suites for exposing XSS vulnerabilities but requires intensive labor as the task of generating mutants is not automated.

3) Vulnerability Detection:

Static-analysis-based approaches avoid vulnerabilities in server-side scripts, but they tend to generate many false positives. Recent approaches combine static analysis with dynamic analysis techniques to improve accuracy. The static analysis identify tainted inputs accessed from external data sources, track the flow of tainted data, and check if any reached sinks such as SQL statements and HTML output statements. An open source vulnerability scanner performs alias analysis to improve accuracy. This is done because sometimes static analysis generate false positives, i.e., a reported vulnerable statement may be actually sanitized by escaping methods. But sometimes it will miss a real vulnerable statement. Static-analysis-based techniques quickly detect potential XSS vulnerabilities in source code and are relatively easy for security personnel to implement and adopt. However, they cannot check the correctness of input sanitization functions and assumes that unhandled or unknown functions return unsafe data. These approaches also miss DOM-based XSS vulnerabilities as they do not target client-side scripts. Thus a combined static and dynamic analysis can be performed to identify faulty sanitization functions.

C. Tool Support

To help developers practice its defensive coding rules, OWASP has created the Enterprise Security API (ESAPI; https://owasp.org/index.php/Category:OWASP Enterprise Security API), an open source library for many different programming languages. Microsoft also provides the Web Protection Library (http://wpl.codeplex.com) for .NET developers. These libraries provide many escaping APIs and other security control features. SecTools maintains a list of the top scanners (http://sectools.org/web-scanners.html). The list includes popular commercial systems such as Acunetix Web Vulnerability Scanner (www.acunetix.com/vulnerabilityscanner) and IBMs Rational AppScan family (http://www-
01.ibm.com/software/awdtools/appscan), as well as open source scanners such as OWASP's WebScarab (https://owasp.org/index.php/Category:OWASP WebScarab Project) and Paros (http://parosproxy.org). All of these scanners generally use either crawlers or proxies to fetch web-pages and then inject predefined attack vectors into response pages, letting users verify the resulting behaviors.

IV. CONCLUSION

The increasing use of the web paradigm for the development of pervasive applications is opening new security threats against the infrastructures behind such applications. Web application's developers must consider the use of support tools to guarantee a deployment free of vulnerabilities, such as secure coding practices, etc. However, attackers continue managing new strategies to exploit web applications. The significance of such attacks can be seen by the pervasive presence of web applications for health care, banking, government administration, and so on. Here we discuss some of the defensive measures which can be deployed in web applications to prevent the SQL Injection and XSS attacks. But each have their own strengths and weaknesses to deal with. Thus an efficient solution to prevent attacks should be enforced to provide security in all environment of web applications. The current problem lies in the integration of these attack prevention techniques in a practical environment and the developers’ familiarities with injection attacks and XSS attacks, and the use of these techniques.

V. REFERENCES


