2015 4th International Conference on Advanced Computer Science Applications and Technologies

Survey of Web Application Vulnerability Attacks

Ossama B. AlKhurafi
Kulliyyah of Information and Communication Technology
International Islamic University Malaysia
Gombak, KL, Malaysia
alkhurafi@yahoo.com

Mohammad A. AlAhmad
Computer Science Department
Public Authority for Applied Education and Training
College of Basic Education
P.O. Box 34567 Adaliyah, 73205 Kuwait City, Kuwait
malahmads@yahoo.com

***Abstract*—Web applications have become an essential part of our daily life. Since web applications contain valuable sensitive information, hackers try to find vulnerabilities and exploit them in order to impersonate the user, steal information, or sabotage the application. This paper illustrates in detail the most prevailing and harmful web application vulnerability attacks: SQL Injection, Broken Authentication and Session Management, and Cross-Site Scripting (XSS).**

Keywords-Web Application Security; Web Vulnerabilities; Web Application Attacks.

I. INTRODUCTION

Web Applications have become a vital part of the web and often contains sensitive data that has to be protected and secured properly. In order to secure a web application, there are three components that need to be secured: integrity of data, confidentiality of data, and availability of web application. This paper will survey the different attacks that compromise these three areas and how the vulnerability of the web application is exploited. However, the focus will be on the most prominent web application vulnerabilities: SQL injection, broken authentication and session management, and cross-site scripting (XSS) attacks.

The Open Web Application Security Project (OWASP) is a non-profit organization that provides un-biased security information regarding web applications. In OWASP's 2010 top 10 most critical web application security risks, the top three web application vulnerability risks used to be injection, cross-site scripting, then followed by broken authentication and session management [1]. However, in OWASP's 2013 top 10, web application vulnerability risks shifted to injection, broken authentication and session management, and then cross-site scripting respectively. This change, according to OWASP, is because the area of broken authentication and session management is being looked at harder, and not because the issues are more prevalent. Their risk methodology is based on evaluating the threat agents, attack vectors, weakness prevalence, weakness detectability, technical impacts, and business impacts of the vulnerability.

II. SQL INJECTION ATTACKS

SQL Injection (SQLi) attack is ranked the most critical web application security risk since it is easy to exploit, widespread, and can have severe impacts on the data of the hosted web application such as data loss or corruption and even complete host takeover. The attacker crafts malicious code in a string that is sent to the database server for execution, which then allows the attacker to steal or manipulate the data from the server. SQL injection attacks occur when the web application sends untrusted data input to the database server, which exploits the syntax of the interpreter. The following are different types of SQL injection attacks:

A. Tautology

The purpose of this attack is to identify injectable parameters, bypass authentication, and extract data [2]. Tautology is a formula that always evaluates to be true. Consider a login page of a web application that asks the user to enter username and password for authentication, and the user input is passed to the database as the SQL query:

SELECT * FROM Accounts WHERE username = 'admin' AND password = 'admin123'

Tautology-based SQL injection can bypass user authentication and extract data by inserting a tautology in the WHERE clause of a SQL query. The attacker, using the tautology technique of SQL injection attacks, can manipulate the query by inserting " ' OR 1 = 1 - - " in the login field of the web application and anything for the password. This results in the following query:

SELECT * FROM Accounts WHERE username = ' ' OR 1 = 1 - - ' AND password = 'anything'

The single quote symbol (') before the OR Boolean Operator indicates the end of the string, and the (- -) symbol is used to comment, which will successfully exit the query. Thus, the query will return true and authenticate the attacker without checking the password.

B. Union Query

The union query injection attack is used to bypass authentication and extract data from the database server. The attacker injects the UNION operator into a vulnerable parameter in the web application that will return the dataset result of both the original and injected queries. For example, consider a business web application connected to a database server that contains an Accounts table to authenticate users and an Employees table with record information of all employees such as name, address, phone number, salary, position, etc. If the login field parameter is vulnerable to SQL injection, the attacker can insert the following

978-1-5090-0424-9/16 $31.00 © 2016 IEEE
DOI 10.1109/ACSAT.2015.46

malicious code injection in the login field “ ' UNION SELECT * FROM Employees - -” and anything for the password, which will result in the following query:

SELECT * FROM Accounts WHERE username = ' ' UNION SELECT * FROM Employees - - ' AND password = 'anything'

The first query will return a null set since there should not be any matching records of an empty username in the Accounts table. However, the second query will return all employee records from the Employees table. Since the UNION operator returns the dataset of both queries, this will enable the attacker to extract the data in the Employees table. Though when combining multiple queries, the number and data types of table columns must be the same. The query can also be crafted differently if the number of columns is different. One technique is to try guessing by injecting “ ' UNION SELECT A,B FROM Employees - -” to check if the tables were two columns, and so forth.

C. Illegal/Logically Incorrect Queries

This can be used as a preliminary attack to identify vulnerable injectable parameters, reveal information about the backend database, and to extract data. By crafting incorrect malicious queries in the input parameter, the attacker can cause the database to return an error message that may contain information about the backend database [3]. This is also referred to as error-based injection. Injected code that creates syntax, type conversion, or logical errors can generate errors that reveal data types of columns and show the names of the tables and columns of the database. Type conversion errors often can show the data type of columns, while logical errors display names of tables and columns. For example, the attacker injects “ ' UNION SELECT SUM(username) FROM Accounts - - ” in the login field of a vulnerable web application to produce the following query:

SELECT * FROM Accounts WHERE username = ' ' UNION SELECT SUM(username) FROM Accounts - - ' AND password = 'anything'

This query will attempt to convert the username column in the Accounts table into an integer. Since this is not a valid type conversion, the database server returns an error message containing information about the database and username column.

D. Stored Procedures

Most databases contain a standard set of stored procedures that extend the functionality of the database and allow for interaction with the operating system. Typical uses for stored procedures are data validation or access control mechanism. Once the attacker discovers which database is in use and the vulnerable injectable parameter, SQLi attacks can be crafted to execute stored procedures provided by that specific database, including procedures that interact with the operating system. For example, queries such as this can be executed by typing “ ' ; SHUTDOWN; - - ” in the username field and anything for the password:

SELECT * FROM Accounts WHERE username = ' ' ; SHUTDOWN; - - ' AND password = 'anything'

This query will cause the database to shutdown. Stored procedures injection attacks can be used for privilege escalation, to execute remote commands, or to perform a denial of service attack such as the previous example.

E. Piggy-Backed Queries

A database is vulnerable to piggy-backed queries if it allows for additional queries to be added to at the end of a valid SQL query. The attacker can exploit this vulnerability to extract or modify data, execute remote commands, and execute a denial of service attack. As shown above, appending the stored procedure “ ' ; SHUTDOWN; - - ” to the query that verifies the username and password cased a denial of service attack. An example of a piggy-backed query that modifies the dataset would be injecting “ ' ; DROP TABLE Employees - -” to the username field and anything for the password, which will result in the following query:

SELECT * FROM Accounts WHERE username = ' ' ; DROP TABLE Employees - - ' AND password = 'anything'

The first query will execute, returning a null set from the Accounts table. When the second piggy-backed query executes, however, it will delete the Employees table from the database.

F. Alternate Encodings

This technique can be used in order to bypass detection methods used by defensive coding practices. Common defensive coding practices scans for and removes harmful code from user input such as “SHUTDOWN” strings that can cause the database to halt. Alternate encodings enable attackers to encode the input string using hexadecimal, ASCII, or Unicode character encoding to conceal the attack, which then will pass undetected and execute.

Consider the code “char(0x73687574646f776e)”, which uses the char() function of ASCII to translate the hexadecimal code 0x73687574646f776e to characters. This will return the string “SHUTDOWN”. Knowing this information, the attacker can craft the following attack “ ' ; exec(char(0x73687574646f776e)); - - ” in the vulnerable username field to produce the following query:

SELECT * FROM Accounts WHERE username = ' ' ; exec(char(0x73687574646f776e)); - - ' AND password = 'anything'

When this query runs, it will pass undetected through the vulnerable parameter and execute the shutdown command, which will cause a denial of service attack. It is difficult to implement a code based defense against this technique since it requires the developer to take into consideration all of the possible encodings that could affect the query string and the database.

G. Inference

This category of attacks is used in order to infer information about more secured databases that do not generate revealing error messages [4]. It can be used to identify injectable parameters and extract data from the database server. The attacker injects queries in the form of true or false statements, and then observes the situation. This technique is also called Blind SQL injection, and it enables

the attacker to test for vulnerabilities even when nothing is returned from the database to the end user. To illustrate this in an example, consider a website with the following URL:

http://onlineshop.com/displayItem.php?id=1

By appending "OR 1 = 1" and then in another instance "AND 1 = 2" to the end of the URL and observing the behavior of the web application, the attacker can find out if that parameter is potentially vulnerable to SQL injection attacks. If the web application returns the same page for "http://onlineshop.com/displayItem.php?id=1 OR 1 = 1" and an error page for "http://onlineshop.com/displayItem.php?id=1 AND 1 = 2" then the web site is potentially vulnerable to SQL injection attacks since both queries were executed as expected.

There is also another inference technique that is called Time-Based SQL injection attack, which relies on manipulating the response time of the database to extract information. The attacker does so by injecting conditional structured queries (containing IF and/or WHEN conditional syntax) and then observes the response time delay. Consider the attacker injects the following in the URL:

http://onlineshop.com/displayItem.php?id=1 AND IF(version() like '4%', sleep(10), 'false'))--

Using this conditional IF statement, the attacker tests to know if the backend database server is MySQL v4.x. If it is true, then a 10 second delay response from the database will be observed (because of the sleep(10) function). If not, then the page will load instantaneously.

III. BROKEN AUTHENTICATION AND SESSION MANAGEMENT

Broken authentication and session management vulnerabilities are widespread and have severe impact once exploited, since it allows the attacker to steal privileged user accounts and hide their actions. The root cause of this vulnerability is that developers often create custom authentication schemes that contain security flaws in areas such as password management, logging out, account update, timeouts, etc. Detecting such flaws can be difficult since each custom authentication and session management implementation is unique.

Consider for example the following web application that exposes the session ID in the URL:

"http://onlineshop.com/item?jsessionid=2D0FN8JPIDJA JDKEIAK5NV"

If the user decides to share this link, any access the user previously had on the web application will also be applicable to the person that visits it. Thus, the attacker can then steal the user's information or perform actions on the web application as the previously authenticated user, such as purchase items from the online shopping site using the victim's credit card information.

Session ID's also need to be generated from the web application's server. The attacker does not need to gain the session ID if the web application is vulnerable to session fixation attacks. In that case, the attacker can create any session ID, email it to the victim or use any technique to trick the user to visit the link, and then follow the same URL to gain access. Also, if session ID's do not properly timeout, it can be used by the attacker at a later time. Lengthy or non-existent session timeouts can also cause users that access the web application through public computers vulnerable to this attack since the attacker can reopen the browser and access the application using the same session ID.

The developers have to follow proper security practices as well, such as encrypting the user's credentials in the database or using a strong hashing function with unique salts for the passwords [5]. This will prevent the attacker from obtaining the account credentials of all users if the database was compromised (ex. SQLi attack). Different security techniques have to be employed as well, such as requiring re-authentication to change user password, and answering security questions for password recovery.

IV. CROSS-SITE SCRIPTING (XSS) ATTACKS

Cross-Site Scripting (XSS) attack is ranked the third most critical web application security risk in OWASP's 2013 top 10 because it is very widespread and can cause a moderate impact, such as execute scripts in a victim's browser to hijack user sessions, deface web sites, insert hostile content, and redirect users to malicious sites. XSS vulnerabilities are injections into the HTML output that the web application generates, which enables the attacker to inject script into web pages viewed by others. This script will then run on the victim's browser, as it came from a trusted source. XSS allow attackers to bypass same-origin policy, which implies that any content from the same site is granted the same permissions, while content from different sites are given permissions separately [6]. The following are the different types of XSS attacks:

A. Reflected (Non-Persistent)

Non-persistent scripts do not reside on the server, and only affects the user running the code directly. It can however indicate that the web site has other vulnerable input parameters. Reflected XSS is harnessed by attaching the malicious script to the end of a link. There are techniques, such as link shortening services, which can disguise the URL. Currently reflected XSS is the most common cross-site scripting method used. The malicious code possibilities are redirect to a phishing site, steal cookie information, and force the user to make an action. Attackers can target a specific user by sending the malicious link through an email, or a large group by publishing the concealed malicious URL on web sites such as social media sites. When a user is tricked into clicking on a malicious link or submitting a specially crafted form, the injected code travels to the vulnerable web server, which reflects the attack back to the user's browser. The browser then executes the code because it came from a "trusted" server.

For instance, the attacker discovers that the web application of a bank contains XSS vulnerability, such as an error page that displays the input, and injects the following script:

"<script> document.location = 'http://www.attackerwebsite.com/logger.php?cookie=' + document.cookie </script>"

The script is appended in the link, and can be crafted to appear as a safe hyperlink. The victim can then be tricked into visiting the malicious link that will redirect the user to the bank's web application, except it will run the hidden script and sends the victim's cookie to the attacker. This cookie can then be used to authenticate the attacker into the bank's website as the victim, steal credit card information, and conduct transactions.

B. Stored (Persistent)

Persistent or stored XSS scripts remain permanently stored on the database, and the attacker's script executes each time a user visits the affected web application. This attack is common in vulnerable web application sites that contain forums, message boards, guest log, comment section, etc. This type of attack can drastically compromise a web application, since the malicious script can be used to steal the victim's cookies that visit the site and send them to the attacker, insert unwanted content such as advertisement, or force users visiting the site to be redirected to the attacker's website. The stored attack can be less harmful such as forcing the web application to display the attacker's advertisement, or more harmful such as stealing the user's cookie. This enables the attacker to impersonate the user since web application's cookies are used to authenticate users and keep the state of the application.

For example, a news forum web application is vulnerable to stored XSS attacks. The attacker submits the following comment:

"Hello <script>alert("XSS")</script>"

This comment will then be stored in the database of the web application, and viewed by each user that visits the web site. If the website is vulnerable to this type of attack, the script will be executed by the browser of users that visits the site without their knowledge, since the comment will be just displayed as "Hello". This script executes on the victim's browser as it is sent from the vulnerable web application, not a different malicious website. This simple script will show a harmless alert popup box with the words "XSS" each time a user loads the page. In order to force redirect the users to another website, the attacker can inject the following java script in the news forum:

"Hello
<script>window.location='http://www.attackerwebsite.com'
</script>"

The attacker in this example uses the java script command "window.location" to force redirect the user's browser that view the vulnerable application to their desired website.

V. RELATED WORK

Li et al. [7] identifies three critical security properties that websites should preserve: input validity, state integrity, and logic correctness. This paper's contribution relies on organizing the research available on securing web applications under three categories, based on their design philosophy: security by construction, security by verification, and security by protection.

Kindy et al. [8] details a survey on SQL injection vulnerabilities, attacks, and prevention techniques. Different types of SQL injection attacks are described, and a comparative analysis is given to base various research works against the different types of SQL injection attacks.

Goswami et al. [9] analyzed the attack surface of web applications to compare the vulnerability reduction of applications that follow OWASP compliance. It was observed that the attack surface has been reduced by 45% once it was made OWASP compliant. The attack surface was defined as the amount of code, functionality, and interfaces of a system exposed to attackers. Heumann et al. [10] method was used to evaluate the attack surface of web applications.

To detect and prevent SQL Injection attacks, AMNESIA [11] uses an approach that involves both static and dynamic run-time monitoring. In the static program analysis phase, it builds models of the different SQL queries the application can generate at every database call point. If the SQL query generated at run-time then violates the model structure, it will be detected and prevented. However, the attack will not be detected if the malicious query matches a legitimate query in a different path from the static model analysis phase.

ADRILLA [12] is an automatic white-box program analysis technique to detect SQL and XSS vulnerabilities in a web application. It is based on tracking tainted data through database access, which is able to detect second-order XSS attacks. However, since it is a white-box program analysis tool, it requires the source code of the application.

VI. CONCLUSION

Web applications became a popular platform that people use daily for shopping, socializing, and banking. However, it attracts hackers as well, since stealing sensitive data can be used for monetary gains. Organized crime became the most frequently seen threat actor for web application attacks, with financial gain being the primary motive. The web application developers have to be aware of the different web application attacks in order to follow proper coding practices. These practices (such as sanitizing user input to prevent injection attacks) can mitigate most widespread web application vulnerabilities.

REFERENCES

[1] Open Web Application Security Project, "OWASP Top Ten Project," https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project/, 2013.

[2] S. Makanadar and V. Solankurkar, "An Approach to Detect and Prevent SQL Injection Attacks using Web Service," International Journal of Science and Research, Vol. 2 Issue 4, pp. 242-245, April 2013.

[3] A. Choudhary and M. Dhore., "CIDT: Detection of Malicious Code Injection Attacks on Web Applications," International Journal of Computer Applications, Vol. 52 No. 2, pp. 19-26, August 2012.

[4] SQL Injection Inference Attacks. http://www.sqlinjection.net/inference/.

[5] M. Jones, “Top Ten Security Risks: Broken Authentication and Session Management,” https://www.credera.com/blog/technology-insights/java/broken-authentication-session-management/, 2014.

[6] A. Doupe, “Advanced Automated Web Application Vulnerability Analysis,” Ph.D. Thesis, University of California Santa Barbara, 2014.

[7] X. Li and Y. Xue, “A Survey on Web Application Security,” Technical Report, Vanderbilt University, http://www.isis.vanderbilt.edu/node/4478/, 2011.

[8] D. Kindy and A. Pathan, “A Survery on SQL Injection: Vulnerabilities, Attack, and Prevention Techniques,” IEEE 15th International Symposium on Consumer Electronics (ISCE), pp. 468-471, 2011.

[9] S. Goswami, N. Krishnan, Mukesh, S. Swarnkar, and P. Mahajan, “Reducing Attack Surface of a Web Application by Open Web Application Security Project Compliance,” Defence Science Journal, Vol. 62 No. 5, pp. 324-330, September 2012.

[10] T. Heumann, S. Turpe, and J. Keller, “Quantifying the Attack Surface of a Web Application,” Proceedings of Sicherheit, Vol. 170, pp. 305-316, 2010.

[11] W. Halfond and A. Orso, “Amnesia: Analysis and Monitoring for Neutralizing SQL-Injection Attacks,” Proceedings of the 20th IEEE and ACM International Conference on Automated Software Engineering, pp. 173-183, 2005.

[12] A. Kiezun, P. Gui, K. Jayaraman, S. Artzi, and M. Ernst, “Automatic Creation of SQL Injection and Cross-Site Scripting Attacks,” Preceedings of the 31th International Conference on Software Engineering, pp. 199-209, 2009.