Automated Web Security Analysis Tool Report

Prepared by Meenakshi Manikandaswamy

I. Executive Summary:

Targeted Web Page: http://sudo.co.il/xss/level0.php

This report details the progress and findings of the Automated Web Security Analysis Tool project, focusing on identifying and mitigating vulnerabilities such as Open Redirection, File Upload, Cross-Site Request Forgery (CSRF), SQL Injection (SQLi), and Cross-Site Scripting (XSS).

II. Open Redirection Vulnerability:

Objective:

An open redirection vulnerability occurs when a web application redirects a user to an untrusted site. It can be exploited by attackers to redirect them to malicious sites, leading to phishing attacks or the theft of sensitive information.

Steps Taken:

Detection:
The tool scans webpages for open redirection vulnerabilities by analyzing URLs and redirects.

Payloads:
Utilizes preset payloads to assess the susceptibility of the webpage to open redirection attacks.
Sample Payload: https://malicious-site.com

Findings:
Number of Attack Vectors Identified: 0
Attack Execution: None Identified

CVSS Score:
Severity: Medium
Score: 6.1

Do It Yourself!

Identify Redirection Points:
Look for any features or parameters in the application that involve redirection. Common examples include login/logout, password reset, or external links.

Understand the Redirection Mechanism:
Analyze how the application handles redirections. Check whether it uses user-supplied data or parameters to determine the target URL.

Provide Malicious Input:
Inject a malicious URL or a URL with a redirect to an external domain. Observe the application's response. If the application redirects to the provided URL without proper validation, there may be a vulnerability.

Explore Redirect Parameters:
If the redirection is controlled by parameters (e.g., redirect or return_url), try manipulating these parameters to introduce variations or inject malicious values.

Check for Filters and Validation:
Investigate if the application has any input validation or filters in place. It may have client-side or server-side filters to prevent malicious redirections.

Prevention:

Whitelisting:
Maintain a whitelist of trusted, allowable redirect URLs within your application. Before redirecting, validate that the supplied URL is present in the whitelist. Reject any redirection attempts to URLs not in the whitelist.

Domain Matching:
Ensure that the redirection URL belongs to the same domain or a trusted domain. This prevents attackers from redirecting users to arbitrary external domains.

Relative URLs:
Whenever possible, use relative URLs for redirection rather than full URLs. This ensures that the redirection stays within the same domain.

III. File Upload Vulnerability:

Objective:

File upload vulnerabilities enable attackers to upload malicious files to a web application, potentially leading to remote code execution, unauthorized access, or the compromise of sensitive data.

Steps Taken:

Detection:
The tool analyzes webpages for file upload elements and assesses their security.

Payloads:
Employs predefined payloads to test the security of file upload functionalities.
Sample Payload: File with a name containing executable code

Findings:
Number of Attack Vectors Identified: 0
Attack Execution: None Identified

CVSS Score:
Severity: High
Score: 8.8

Do It Yourself!

Upload Malicious File:
Attempt to upload a file that contains malicious code, such as a web shell or a file with embedded scripts.

Check for File Extension Bypass:
Some applications validate files based on file extensions. Try to bypass this by renaming a file with a valid extension but containing malicious code (e.g., renaming a .php file to .jpg).

Check for Client-Side Validation:
Use browser developer tools to inspect and modify the HTML form, disabling any client-side validation that may be present.

Bypass Frontend Validation:
If there is client-side validation, try to bypass it by manipulating the upload request using tools like Burp Suite or modifying the HTML form.

Explore File Upload Permissions:
Investigate how the application handles file permissions after upload. Ensure that uploaded files cannot be accessed or executed by unauthorized users.

Prevention:

File Type Validation:
Verify the file type by checking the file extension or using file signature analysis. Ensure that only allowed file types are accepted. Do not rely solely on client-side checks; perform server-side validation as well.

Content-Type Header Check:
Validate the Content-Type header of the file to ensure it matches the expected type. This helps prevent attackers from manipulating file extensions.

Disable Execution of Uploaded Files:
Ensure that uploaded files are not executable. Store uploaded files in a location separate from executable scripts and prevent the execution of uploaded content.

File Size Limitation:
Set a maximum file size limit to prevent the upload of excessively large files, which can lead to denial-of-service (DoS) attacks.

Secure File Storage:
Store uploaded files outside the web root directory to prevent direct access.
Use proper access controls to restrict who can access and modify uploaded files.

Further Reading:

https://portswigger.net/web-security/file-upload
https://owasp.org/www-community/vulnerabilities/Unrestricted_File_Upload
https://cheatsheetseries.owasp.org/cheatsheets/File_Upload_Cheat_Sheet.html
https://cwe.mitre.org/data/definitions/434.html

IV. Cross Site Request Forgery (CSRF):

Objective:

CSRF occurs when an attacker tricks a user's browser into making unintended and unauthorized requests on behalf of the user. This can lead to actions being performed on the web application without the user's consent.

Steps Taken:

Detection:
Identifying CSRF tokens and analyzes form elements for potential vulnerabilities.

Payloads:
Uses predefined payloads to simulate CSRF attacks and assess vulnerability.
Sample Payload: Tricking user into submitting a form with malicious content.

Findings:
Number of Attack Vectors Identified: 1
Attack Execution: None Identified

CVSS Score:
Severity: Medium
Score: 6.5

Do It Yourself!

Analyze Form Submissions:
Identify forms within the application that perform actions with side effects and those that lack anti-CSRF mechanisms like tokens or same-site cookie attributes.

Craft a Malicious Page:
Create a malicious HTML page that includes a hidden form with the target action and parameters.
Ensure that the form mimics the structure of the legitimate forms in the application.

Host the Malicious Page:
Host the malicious HTML page on an external server accessible by the victim.
The page should contain JavaScript to automatically submit the form.

Trick the Victim:
Convince the victim to visit the malicious page, often through social engineering tactics like phishing emails or disguised links.
Monitor the server logs to see if the forged request was processed successfully.

Prevention:

Use Anti-CSRF Tokens:
Include unique, unpredictable tokens in each HTML form. These tokens should be generated on the server side and embedded in the form as hidden fields or headers. Upon form submission, the server validates the token to ensure that the request originated from the legitimate application.

SameSite Cookie Attribute:
Set the SameSite attribute for cookies to mitigate the risk of CSRF attacks. This attribute restricts when the browser sends cookies, ensuring they are only sent in requests originating from the same site as the target URL.
Use SameSite=Lax or SameSite=Strict to provide varying levels of protection.

Check Referrer Header:
Verify the Referer header on the server to ensure that requests originate from the expected domain. However, note that the Referer header can be manipulated or omitted in some cases.

Custom Headers:
Include custom headers in requests and validate them on the server. Ensure that these headers are not accessible to scripts running in the browser (e.g., using the Access-Control-Expose-Headers header).

Implement Time-Based Tokens:
Include a timestamp in the token and validate it on the server. Reject requests with expired tokens.

Further Reading:

https://portswigger.net/web-security/csrf
https://owasp.org/www-community/attacks/csrf
https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention
https://cwe.mitre.org/data/definitions/352.html
https://csrc.nist.gov/glossary/term/cross_site_request_forgery
https://portswigger.net/web-security/csrf/preventing

V. SQL Injection (SQLi):

Objective:

SQL injection vulnerabilities allow attackers to execute arbitrary SQL queries on a web application's database. This can lead to unauthorized access, data manipulation, or even the deletion of the entire database.

Steps Taken:

Detection:
The tool analyzes input fields for potential SQL injection points.

Payloads:
Employs predefined SQL injection payloads to assess the security of input fields.
Sample Payload: Inputting (' OR '1'='1';) -- to bypass a login form.

Findings:
Number of Attack Vectors Identified: 1
Attack Execution: !!ATTACK SUCCEEDED!!

CVSS Score:
Severity: High
Score: 8.8

Do It Yourself!

Identify Input Fields:
Locate input fields or parameters where user input is processed, such as login forms, search boxes, or URL parameters.

Input Malicious SQL Payloads:
Inject malicious SQL payloads into the input fields to exploit potential vulnerabilities

Error-Based SQL Injection:
Intentionally trigger SQL errors by injecting payloads like
' OR 1=CONVERT(int, (SELECT @@version)); --.
Observe error messages that may reveal information about the database.

Time-Based Blind SQL Injection:
Use time-delayed payloads to detect if the application is vulnerable to blind SQLi
An example payload is ' OR IF(1=1, SLEEP(5), 0); --.

Boolean-Based Blind SQL Injection:
Craft payloads that rely on true/false conditions to extract information.
Example payload: ' OR 1=1 --.

UNION-Based SQL Injection:
Exploit the UNION SQL operator to combine the results with additional information.
Example payload: ' UNION SELECT username, password FROM users; --.

Out-of-Band SQL Injection:
Utilize techniques like DNS requests or HTTP requests to retrieve data when classic SQL injection methods are restricted.

Prevention:

Parameterized Queries or Prepared Statements:
Instead of directly embedding user input into SQL queries, use parameterized queries or prepared statements provided by the programming language or database library. These mechanisms separate user input from the SQL query structure.

Avoid Dynamic SQL Construction:
Avoid dynamically constructing SQL queries by concatenating strings with user input. Dynamic SQL is prone to injection attacks.

Input Validation:
Validate and sanitize user inputs before using them in SQL queries. Input validation ensures that the input adheres to expected patterns, reducing the risk of SQLi.

Use Stored Procedures:
Employ stored procedures to encapsulate and execute database logic.
Stored procedures help mitigate SQL injection by predefining the SQL operations that can be performed.

Escaping Special Characters:
Escape or sanitize special characters in user input to neutralize their potential impact on SQL queries.

VI. Cross-Site Scripting (XSS):

Objective:

XSS vulnerabilities enable attackers to inject malicious scripts into web pages, which are then executed by users' browsers. This can lead to the theft of sensitive information, session hijacking, or defacement of websites.

Steps Taken:

Detection:
The tool scans webpages for input fields and text elements susceptible to XSS attacks.

Payloads:
Utilizes predefined XSS payloads to assess and simulate XSS attacks.
Sample Payload: Injecting <script>alert('XSS')</script> in an input field.

Findings:
Number of Attack Vectors Identified: 2
Attack Execution: !!ATTACK SUCCEEDED!!

CVSS Score:
Severity: Medium
Score: 6.1

Do It Yourself!

Identify Input Fields:
Locate input fields on web pages such as search bars, text boxes, and form fields.

Test with Attributes:
Inject payloads into HTML attributes. Examples include:
<img src=x onerror=alert('XSS')>
Click me

Test with Malformed HTML:
Use payloads that exploit browser parsing issues. Examples include:
<b onmouseover=alert('XSS')>mouseover here

Test Event Handlers:
Check event handlers for vulnerabilities. Examples include:
onmouseover=`alert('XSS')`

Test URL Parameters:
Inject payloads into URL parameters. Examples include:
?search=<script>alert('XSS')</script>
?name=<img src=x onerror=alert('XSS')>

Test for Delayed Execution:
Use payloads that might be triggered after a delay, making detection challenging.
<script>setTimeout(function(){alert('XSS')}, 5000);</script>

Test for Stored XSS:
If applicable, check for stored XSS vulnerabilities by injecting payloads that are stored and later displayed to other users.

Prevention:

Input Validation:
Validate and sanitize all user inputs on both client and server sides.
Define and enforce a strict whitelist of allowed characters, rejecting any input that doesn't adhere to the defined criteria.

Output Encoding:
Encode user-generated content before rendering it in the HTML document.
This involves converting special characters to their HTML entities.
Use proper output encoding functions such as htmlspecialchars() in PHP, escapeHtml in Java, or equivalent functions in other programming languages.

Content Security Policy (CSP):
Implement Content Security Policy headers in your web application.
CSP helps control which resources can be loaded and executed, reducing the risk of executing malicious scripts.
Configure CSP to block or restrict the use of inline scripts and unsafe practices.

HTTP-Only Cookies:
Set the HTTP-only flag on cookies to prevent JavaScript access. This ensures that sensitive information, such as session cookies, cannot be accessed by malicious scripts.

Contextual Output Encoding:
Encode output based on the context where it will be used. Different contexts (HTML, attribute, JavaScript, etc.) have different encoding requirements.
Use functions specific to the output context, such as htmlspecialchars for HTML content and encodeURI for JavaScript.

Avoiding eval():
Avoid using eval() and other dynamic code execution functions.
If dynamic code execution is necessary, validate and sanitize inputs thoroughly before execution.

Secure Headers:
Implement secure headers such as X-Content-Type-Options and X-Frame-Options to control content types and framing policies.

Use Security Libraries:
Leverage security libraries and frameworks that provide built-in protection against XSS, such as Java's OWASP AntiSamy or Ruby on Rails' sanitize helper.

Regular Expressions:
Use regular expressions to filter and sanitize input. Ensure that input adheres to expected patterns and reject any input that doesn't meet the criteria.

Further Reading:

https://portswigger.net/web-security/cross-site-scripting
https://owasp.org/www-community/attacks/xss/
https://csrc.nist.gov/glossary/term/cross_site_scripting
https://owasp.org/www-community/Types_of_Cross-Site_Scripting
https://portswigger.net/web-security/cross-site-scripting/reflected
https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat
https://owasp.org/www-project-web-security-testing-guide/latest/4-Web_Application_
https://owasp.org/www-project-top-ten/2017/A7_2017-Cross-Site_Scripting_(XSS)
https://cheatsheetseries.owasp.org/cheatsheets/XSS_Filter_Evasion_Cheat_Sheet.html
https://ieeexplore.ieee.org/document/10128470
https://owasp.org/www-community/attacks/DOM_Based_XSS
https://cheatsheetseries.owasp.org/cheatsheets/DOM_based_XSS_Prevention_Cheat_S

VII. Reference Links:

To provide comprehensive support, the tool includes reference links and step-by-step guides for each vulnerability. These resources aid users in understanding the nature of the vulnerabilities detected and offer recommendations for mitigation.

Additional references for Open Redirection Vulnerability:

https://cwe.mitre.org/data/definitions/601.html
https://portswigger.net/kb/issues/00500100_open-redirection-reflected
https://cheatsheetseries.owasp.org/cheatsheets/Unvalidated_Redirects_and_Forwards_
https://owasp.org/www-project-web-security-testing-guide/v41/4-Web_Application_Se
https://portswigger.net/support/using-burp-to-test-for-open-redirections
https://nvd.nist.gov/vuln/detail/CVE-2023-22797

Additional references for SQL Injection Vulnerability:

https://owasp.org/www-community/attacks/SQL_Injection
https://portswigger.net/web-security/sql-injection
https://portswigger.net/kb/issues/00100200_sql-injection
https://owasp.org/www-project-web-security-testing-guide/latest/4-Web_Application_
https://capec.mitre.org/data/definitions/66.html
https://www.cisa.gov/sites/default/files/publications/Practical-SQLi-Identification.pdf

VIII. Conclusion:

The Automated Web Security Analysis Tool serves as a crucial asset in identifying and addressing common web application vulnerabilities. As the project evolves, further enhancements will be made to maximize the tool's effectiveness and usability, ensuring its reliability across diverse web development environments.

Note: CVSS scores will need to be determined based on the specific findings and impact of each vulnerability. Consult the Common Vulnerability Scoring System (CVSS) for accurate scoring.