

Web Application Security Assessment

Prepared by DeepDefence Technologies

Target Domain: http://app.example.com

Generated On: Fri, 19 Sept 2025 09:10:22

Executive Summary

STATUS: IMMEDIATE ATTENTION REQUIRED

The application is currently vulnerable to high-risk exploits. These vulnerabilities could allow attackers to bypass authentication or extract sensitive database information.

Severity Summary

Risk Level	Total Findings
High	3
Medium	0
Low	0
Informational	3

Detailed Vulnerability Findings

Vulnerability	Risk Level	Instances	CWE ID
Cross Site Scripting (Reflected)	High (Medium)	2	CWE-79
SQL Injection	High (Medium)	1	CWE-89
SQL Injection - MySQL	High (Medium)	3	CWE-89
Modern Web Application	Informational (Medium)	1	N/A
Session Management Response Identified	Informational (Medium)	7	N/A
User Controllable HTML Element Attribute (Potential XSS)	Informational (Low)	1	CWE-20

Technical Breakdown & Strategy

1. Cross Site Scripting (Reflected)

Description: Cross-site Scripting (XSS) is an attack technique that involves echoing attacker-supplied code into a user's browser instance. A browser instance can be a standard web browser client, or a browser object embedded in a software product such as the browser within WinAmp, an RSS reader, or an email client. The code itself is usually written in HTML/JavaScript, but may also extend to VBScript, ActiveX, Java, Flash, or any other browser-supported technology. When an attacker gets a user's browser to execute his/her code, the code will run within the security context (or zone) of the hosting web site. With this level of privilege, the code has the ability to read, modify and transmit any sensitive data accessible by the browser. A Cross-site Scripted user could have his/her account hijacked (cookie theft), their browser redirected to another location, or possibly shown fraudulent content delivered by the web site they are visiting. Cross-site Scripting attacks essentially compromise the trust relationship between a user and the web site. Applications utilizing browser object instances which load content from the file system may execute code under the local machine zone allowing for system compromise. There are three types of Cross-site Scripting attacks: non-persistent, persistent and DOM-based. Non-persistent attacks and DOM-based attacks require a user to either visit a specially crafted link laced with malicious code, or visit a malicious web page containing a web form, which when posted to the vulnerable site, will mount the attack. Using a malicious form will oftentimes take place when the vulnerable resource only accepts HTTP POST requests. In such a case, the form can be submitted automatically, without the victim's knowledge (e.g. by using JavaScript). Upon clicking on the malicious link or submitting the malicious form, the XSS payload will get echoed back and will get interpreted by the user's browser and execute. Another technique to send almost arbitrary requests (GET and POST) is by using an embedded client, such as Adobe Flash. Persistent attacks occur when the malicious code is submitted to a web site where it's stored for a period of time. Examples of an attacker's favorite targets often include message board posts, web mail messages, and web chat software. The unsuspecting user is not required to interact with any additional site/link (e.g. an attacker site or a malicious link sent via email), just simply view the web page containing the code.

Top Occurrences:

- [GET] http://app.example.com/vulnerabilities/view_source.php?id=%3C%2Fh2%3E%3Cscript%3Ealert%281%29%3B%3C%2Fscript%3E%3Ch2%3E&security=impossible
- [GET] http://app.example.com/vulnerabilities/view_source.php?id=open_redirect&security=%3C%2Fh2%3E%3Cscript%3Ealert%281%29%3B%3C%2Fscript%3E%3Ch2%3E

Recommended Strategy:

Recommended Strategy: Phase: Architecture and Design Use a vetted library or framework that does not allow this weakness to occur or provides constructs that make this weakness easier to avoid. Examples of libraries and frameworks that make it easier to generate properly encoded output include Microsoft's Anti-XSS library, the OWASP ESAPI Encoding module, and Apache Wicket. Phases: Implementation; Architecture and Design Understand the context in which your data will be used and the encoding that will be expected. This is especially important when transmitting data between different components, or when generating outputs that can contain multiple encodings at the same time, such as web pages or multi-part mail messages. Study all expected communication protocols and data representations to determine the required encoding strategies. For any data that will be output to another web page, especially any data that was received from external inputs, use the appropriate encoding on all non-alphanumeric characters. Consult the XSS Prevention Cheat Sheet for more details on the types of encoding and escaping that are needed. Phase: Architecture and Design For any security checks that are performed on the client side, ensure that these checks are duplicated on the server side, in order to avoid CWE-602. Attackers can bypass the client-side checks by modifying values after the checks have been performed, or by changing the client to remove the client-side checks entirely. Then, these modified values would be submitted to the server. If available, use structured mechanisms that automatically enforce the separation between data and code. These mechanisms may be able to provide the relevant quoting, encoding, and validation automatically, instead of relying on the developer to provide this capability at every point where output is generated. Phase: Implementation For every web page that is generated, use and specify a character encoding such as ISO-8859-1 or UTF-8. When an encoding is not specified, the web browser may choose a different encoding by guessing which encoding is actually being used by the web page. This can cause the web browser to treat certain sequences as special, opening up the client to subtle XSS attacks. See CWE-116 for more mitigations related to encoding/escaping. To help mitigate XSS attacks against the user's session cookie, set the session cookie to be HttpOnly. In browsers that support the HttpOnly feature (such as more recent versions of Internet Explorer and Firefox), this attribute can prevent the user's session cookie from being accessible to malicious client-side scripts that use document.cookie. This is not a complete solution, since HttpOnly is not supported by all browsers. More importantly, XMLHttpRequest and other powerful browser technologies provide read access to HTTP headers, including the Set-Cookie header in which the HttpOnly flag is set. Assume all input is malicious. Use an "accept known good" input validation strategy, i.e., use an allow list of acceptable inputs that strictly conform to specifications. Reject any input that does not strictly conform to specifications, or transform it into something that does. Do not rely exclusively on looking for malicious or malformed inputs (i.e., do not rely on a deny list).

2. SQL Injection

Description: SQL injection may be possible.

Top Occurrences:

- [GET] http://app.example.com/vulnerabilities/xss_d/?default=French%27+AND+%271%27%3D%271%27+--+

Recommended Strategy:

Recommended Strategy: Do not trust client side input, even if there is client side validation in place. In general, type check all data on the server side. If the application uses JDBC, use PreparedStatement or CallableStatement, with parameters passed by '?'. If the application uses ASP, use ADO Command Objects with strong type checking and parameterized queries. If database Stored Procedures can be used, use them. Do not concatenate strings into queries in the stored procedure, or use 'exec', 'exec immediate', or equivalent functionality! Do not create dynamic SQL queries using simple string concatenation. Escape all data received from the client. Apply an 'allow list' of allowed characters, or a 'deny list' of disallowed characters in user input. Apply the principle of least privilege by using the least privileged database user possible. In particular, avoid using the 'sa' or 'db-owner' database users. This does not eliminate SQL injection, but minimizes its impact. Grant the minimum database access that is necessary for the application.

3. SQL Injection - MySQL

Description: SQL injection may be possible.

Top Occurrences:

- [POST] http://app.example.com/vulnerabilities/authbypass/change_user_details.php
- [POST] http://app.example.com/vulnerabilities/authbypass/change_user_details.php
- [POST] http://app.example.com/vulnerabilities/authbypass/change_user_details.php

Recommended Strategy:

Recommended Strategy: Do not trust client side input, even if there is client side validation in place. In general, type check all data on the server side. If the application uses JDBC, use PreparedStatement or CallableStatement, with parameters passed by '?'. If the application uses ASP, use ADO Command Objects with strong type checking and parameterized queries. If database Stored Procedures can be used, use them. Do not concatenate strings into queries in the stored procedure, or use 'exec', 'exec immediate', or equivalent functionality! Do not create dynamic SQL queries using simple string concatenation. Escape all data received from the client. Apply an 'allow list' of allowed characters, or a 'deny list' of disallowed characters in user input. Apply the principle of least privilege by using the least privileged database user possible. In particular, avoid using the 'sa' or 'db-owner' database users. This does not eliminate SQL injection, but minimizes its impact. Grant the minimum database access that is necessary for the application.

4. Modern Web Application

Description: The application appears to be a modern web application. If you need to explore it automatically then the Ajax Spider may well be more effective than the standard one.

Top Occurrences:

- [GET] <http://app.example.com>

Recommended Strategy:

Recommended Strategy: This is an informational alert and so no changes are required.

5. Session Management Response Identified

Description: The given response has been identified as containing a session management token. The 'Other Info' field contains a set of header tokens that can be used in the Header Based Session Management Method. If the request is in a context which has a Session Management Method set to "Auto-Detect" then this rule will change the session management to use the tokens identified.

Top Occurrences:

- [GET] <http://app.example.com/login.php>
- [POST] <http://app.example.com/login.php>
- [POST] <http://app.example.com/vulnerabilities/brute/>

Recommended Strategy:

Recommended Strategy: This is an informational alert rather than a vulnerability and so there is nothing to fix.

6. User Controllable HTML Element Attribute (Potential XSS)

Description: This check looks at user-supplied input in query string parameters and POST data to identify where certain HTML attribute values might be controlled. This provides hot-spot detection for XSS (cross-site scripting) that will require further review by a security analyst to determine exploitability.

Top Occurrences:

- [GET] <http://app.example.com/instructions.php?doc=readme>

Recommended Strategy:

Recommended Strategy: Validate all input and sanitize output it before writing to any HTML attributes.
