Case Studies on Web Application Vulnerabilities: Analyzing the Causes, Impact and Mitigation Strategies

Akhil Patel
2nd Year CSE, School of Computing
MIT-ADT University
Pune, India
ADT24SOCB0111@mituniversity.edu.in

Atharva Kumar
2nd Year CSE, School of Computing
MIT-ADT University
Pune, India
ADT24SOCB0272@mituniversity.edu.in

Divy Gupta
2nd Year CSE, School of Computing
MIT-ADT University
Pune, India
ADT24SOCB0382@mituniversity.edu.in

Tanish Shetty
2nd Year CSE, School of Computing
MIT-ADT University
Pune, India
ADT24SOCB1273@mituniversity.edu.in

Abstract—As the use of web applications rises, so does the risk associated with them. This paper is a collection of case study analysis of real-world cases of web application vulnerabilities, examining the causes, impact, and potential mitigation strategies. We analyzed five different case studies with a range of security flaws, including OS command injection, CORS misconfiguration, SQL injection (SQLi), cross-site scripting (XSS), broken access control, HTML injection, web cache poisoning, and more. We aim to shed light into these real-world cases of web exploitation, to emphasize the need for proper implementation of security practices in web development. We also focused on comparing these cases on the OWASP ranking to show the prevalence of the common vulnerabilities.

Keywords—*Cybersecurity; Web Application Security; Case Study; Penetration Testing; Web Vulnerabilities.*

I. INTRODUCTION

Over the past decade, web applications have become deeply embedded in how industries operate—from banking to healthcare to government services. While this shift has enabled unprecedented digital connectivity, it has simultaneously expanded the opportunities for malicious actors to exploit weaknesses in these systems. Businesses, individuals, and government entities are all increasingly at risk as web applications continue to serve as prime targets for cyberattacks, often harboring a broad range of security weaknesses and configuration flaws **[4]**.

To counter these escalating threats, security professionals and developers worldwide have worked together to establish standardized guidelines for building safer systems. Among the most prominent contributors to this effort is the Open Worldwide Application Security Project (OWASP), a non-profit organization dedicated to improving software security **[1]**. OWASP regularly publishes research identifying and ranking the most commonly exploited vulnerabilities in web applications **[1]**, serving as a critical reference point for developers and security teams alike.

This research paper examines five real-world web application security scenarios and maps them to OWASP's most prevalent vulnerability categories **[1]**. The study analyzes root causes, impacts, and mitigation strategies for each case, with the goal of providing practical insights for security practitioners and developers **[7]**.

II. CASE STUDIES

A. CORS Misconfiguration Leading to Account Takeover: WA1

1) Introduction:

CORS is a browser-enforced policy that determines whether a web page hosted on one origin is permitted to request data from a server on a different origin **[1]**. Servers communicate their sharing preferences through specific HTTP response headers, most notably Access-Control-Allow-Origin, which specifies which external domains may receive the server's responses. When this policy is poorly configured, it can inadvertently grant unauthorized parties access to sensitive application data **[5]**.

This case study examines a CORS misconfiguration in WA1 that resulted in arbitrary origin access, allowing unauthorized websites to read API responses. The misconfiguration was exacerbated by the inclusion of authentication cookies in the response body whenever session expiration occurred, allowing attackers to perform session hijacking and full account takeover.

2) Vulnerability Analysis:

During security testing of WA1, the server had the following CORS policy header: Access-Control-Allow-Origin: https://web-application.*. The wildcard (*) at the subdomain level resulted in an overly permissive policy **[5]**, allowing any domain matching https://web-application.<TLD> to access server responses, permitting attackers hosting https://web-application.online to gain access to user data.

Additionally, WA1 contained a session management flaw wherein, upon session expiration, the server reflected new authentication cookies in the response body rather than setting them securely via Set-Cookie headers. An attacker exploiting the CORS misconfiguration could extract and reuse these cookies for account takeover.

3) Exploitation Methodology:

A malicious domain was registered: https://web-application.online. The attacker-controlled website hosted a JavaScript payload which, upon tricking a victim into visiting the malicious domain, caused the victim's browser to automatically include authentication cookies in the request. Due to the CORS misconfiguration, the response—including newly generated session cookies—was leaked to the attacker.

4) Mitigation Strategies:

• Whitelist only known, explicitly verified domains in the Access-Control-Allow-Origin header (e.g., https://web-application.com). Partial wildcards and open wildcard values introduce unpredictable trust boundaries and must be avoided entirely **[6]**.

• Avoid exposing authentication tokens in response bodies. Use secure Set-Cookie headers with HttpOnly and SameSite attributes.

• Implement Content Security Policy (CSP) headers to restrict JavaScript execution on unauthorized domains **[6]**.

5) Conclusion:

This case study highlights the severe impact of CORS misconfigurations when combined with improper session management practices **[1]**. The findings underscore the importance of secure access control policies to prevent unauthorized data leakage through CORS vulnerabilities **[5]**.

B. OS Command Injection Leading to Remote Code Execution: WA2

1) Introduction:

When web applications pass user-controlled data directly to underlying operating system functions without adequate filtering, attackers can inject arbitrary shell commands that execute with the server's privileges—a vulnerability class known as OS command injection **[3]**. In worst-case scenarios, this grants full remote code execution (RCE), effectively placing the entire server under the attacker's control. WA2 exhibited precisely this weakness: insufficient input filtering in its search functionality, compounded by an exposed developer testing directory, enabled an attacker to obtain an interactive shell as the www-data user.

2) Vulnerability Analysis:

Two major security flaws were identified: unsanitized user input in a search functionality allowing direct execution of OS commands **[3]**; and broken access control exposing a /testing directory containing an unprotected developer debug tool **[2]**. The application dynamically processed search inputs by concatenating user-supplied data into a system command via shell_exec(). The payload: search=; whoami returned www-data, confirming successful OS command execution.

3) Mitigation Strategies:

• Replace dynamic command construction with parameterized alternatives and enforce whitelist-based input validation **[7]**. Raw user input must never reach a shell execution function under any circumstance.

• Apply the principle of least privilege **[6]**: the www-data process account should only have permissions strictly required to serve web content, limiting the damage an attacker can cause after exploitation.

• Disable dangerous binaries (nc, wget, curl) in web environments and secure all testing directories with authentication.

4) Conclusion:

This case study highlights the dangers of OS command injection **[3]** and how improper input validation combined with exposed testing environments can lead to full system compromise.

C. Blind Time-Based SQL Injection: WA3

1) Introduction:

SQL injection attacks exploit the practice of embedding unfiltered user input directly into database query strings **[3]**. By injecting specially crafted SQL syntax, an attacker can alter the intended logic of the query, potentially bypassing authentication, reading confidential records, or corrupting stored data **[5]**. In WA3's login portal at race.wa3.com, the absence of input sanitization created exactly this opening, enabling an attacker to manipulate the authentication query through injected SQL commands.

2) Vulnerability Analysis:

The application did not return visible database errors, indicating blind SQL injection. A time-based payload was used: \' OR IF(1=1, SLEEP(20), 0) --. This instructs the database to pause for 20 seconds if the condition evaluates successfully. The web application's response time increased to 20,000ms, confirming successful SQL injection **[3]**. This vulnerability could enable authentication bypass, data extraction, and database integrity threats.

3) Mitigation Strategies:

- Adopt prepared statements and parameterized queries throughout all database interactions **[7]**. This approach treats user-supplied values as data rather than executable code, structurally eliminating SQL injection as a threat vector.
- Enforce strict input validation: restrict length and format, reject special characters like \' -- ; unless required.
- Implement a Web Application Firewall (WAF) to detect and block common injection patterns **[6]**.

4) Conclusion:

Failure to sanitize user input can allow attackers to manipulate authentication processes, execute unauthorized SQL queries, and compromise sensitive data **[3]**. The findings underscore the importance of parameterized queries and security-aware development practices **[7]**.

D. Broken Access Control via Improper Redirection: WA4

1) Introduction:

Access control failures occur when an application does not properly verify whether a user is permitted to perform a requested action or view a requested resource before fulfilling the request **[2]**. Rather than proactively blocking unauthorized access, WA4's implementation attempted to redirect unauthenticated users after already serving restricted content—a client-side control that attackers can trivially bypass, leading to unauthorized admin panel access and potential privilege escalation.

2) Vulnerability Analysis:

Two critical weaknesses were identified **[2]**: (1) Weak Authorization Workflow—the application granted access to the admin panel before checking the user's authorization status, relying on client-side redirection if authentication failed; and (2) Improper HTTP Response Handling—the /admin panel returned 200 OK instead of 403 Forbidden. An attacker intercepting the request via Burp Suite **[8]** could prevent the redirection script from executing, viewing and modifying admin content.

3) Mitigation Strategies:

- Authorization decisions must be made and enforced entirely on the server side, prior to returning any content **[6]**. Relying on client-side scripts or post-response redirects to block unauthorized users is fundamentally insecure and trivially bypassed using proxy tools **[8]**.
- Return HTTP 403 Forbidden for unauthorized access attempts instead of 200 OK with subsequent redirections.
- Adopt Role-Based Access Control (RBAC) to systematically restrict access to sensitive administrative interfaces **[2]**.

4) Conclusion:

The failure of WA4 to enforce authentication checks before serving restricted content allowed unauthorized users to bypass authorization controls and gain administrative privileges **[2]**. Security audits should be conducted regularly to ensure authentication and authorization workflows function as intended **[6]**.

E. Web Cache Poisoning via Improper Caching: WA5

1) Introduction:

Web caching systems improve application performance by storing server responses and replaying them to subsequent users without re-processing the original request. This efficiency, however, becomes a liability when the cache stores responses that have been influenced by attacker-controlled inputs **[5]**. In WA5, the caching layer incorporated the X-Origin-Accepted-Language header into its stored responses without first validating its contents—providing an attacker a pathway to inject malicious values that would then be served to unsuspecting users.

2) Vulnerability Analysis:

The caching mechanism did not validate request headers before storing responses **[5]**. The server stored responses based on the X-Origin-Accepted-Language value without validation, allowing arbitrary input to be stored. An attacker sent a modified request containing an evil-payload header value, causing the server to cache the attacker-controlled response. Future users then received this poisoned response. Severity was moderate due to limited header reachability and non-critical header type.

3) Mitigation Strategies:

• Audit and restrict which request headers are permitted to influence cache key generation **[7]**. Any header that is not explicitly validated and expected should be stripped before it reaches the caching layer, ensuring attacker-supplied values cannot be persisted in cached responses.

• Validate and sanitize all input headers before processing, especially those included in cache key computation.

• Implement cache control policies to prevent caching of sensitive pages or authenticated responses **[6]**.

4) Conclusion:

While this vulnerability's severity was low, improper caching of more sensitive headers could lead to session hijacking, content injection, and large-scale cache poisoning attacks **[5]**.

III. FINDINGS

The analysis of vulnerabilities across five case studies reveals that injection attacks were the most prevalent, occurring in four out of five cases (80%)—specifically in WA2, WA3, WA4, and WA5. This aligns with OWASP's 2021 Top 10 ranking, where Injection (A03) remains one of the most critical security risks **[1]**.

Broken Access Control (A01) was identified in three out of five cases (60%)—WA2, WA3, and WA4. This is consistent with OWASP's 2021 list, where Broken Access Control was ranked as the number one most critical vulnerability due to the severe impact of unauthorized access and privilege escalation **[1]**, **[2]**.

Taken together, these results indicate that fundamental security gaps—particularly around input handling and authorization enforcement—remain widespread in real-world web applications **[4]**. The strong correlation between our findings and OWASP's global vulnerability rankings **[1]** suggests these are not isolated incidents but deeply rooted, industry-wide challenges that demand sustained attention from the development community.

TABLE I. Comparison of Vulnerabilities Found in WA1–WA5

Case	Vulnerabilities	Severity
WA1	1. CORS Misconfiguration 2. Auth Cookie Reflection	1× High, 1× Low
WA2	1. OS Command Injection 2. SQL Injection 3. HTML Injection 4. Broken Access Control	1× Critical, 3× High
WA3	1. SQL Injection (SQLi) 2. Broken Access Control	2× High
WA4	1. Broken Access Control 2. Reflected XSS	2× High
WA5	1. Cache Poisoning	1× Medium

TABLE II. Mapping Vulnerabilities to OWASP Top 10

Case	Vulnerabilities Found	OWASP Mapping [1]
WA1	CORS Misconfiguration, Auth Cookie Reflection	A05 – Security Misconfiguration
WA2	OS Command Injection, SQL Injection, HTML Injection, Broken Access Control	A03 – Injection A01 – Broken Access Control
WA3	SQL Injection, Broken Access Control	A03 – Injection A01 – Broken Access Control
WA4	Broken Access Control, Reflected XSS	A01 – Broken Access Control A03 – Injection
WA5	Web Cache Poisoning	A05 – Security Misconfiguration

IV. CONCLUSION

This study highlights that injection attacks (80%) and broken access control vulnerabilities (60%) remain critical threats, aligning with OWASP's security rankings **[1]**, **[2]**, **[3]**. These results emphasize the necessity of stricter input validation and access control mechanisms **[6]**, **[7]** to protect modern web applications.

Future research could expand this study by analyzing a broader set of web applications across different industries, as well as evaluating the effectiveness of modern mitigation techniques such as AI-driven threat detection and automated vulnerability scanning frameworks **[4]**, **[8]**.

REFERENCES

[1] OWASP Foundation, "OWASP Top Ten 2021," 2021. [Online]. Available: https://owasp.org/www-project-top-ten/ [Accessed: May 2024].

[2] OWASP Foundation, "Broken Access Control," OWASP Top 10, 2021. [Online]. Available: https://owasp.org/www-project-broken-access-control/ [Accessed: May 2024].

[3] OWASP Foundation, "Injection Attacks," OWASP Community Pages, 2021. [Online]. Available: https://owasp.org/www-community/attacks/ [Accessed: May 2024].

[4] Verizon, "2023 Data Breach Investigations Report (DBIR)," Verizon Business, Basking Ridge, NJ, USA, Tech. Rep., 2023. [Online]. Available: https://www.verizon.com/business/resources/reports/dbir/

[5] MITRE Corporation, "Common Weakness Enumeration (CWE) Database," 2023. [Online]. Available: https://cwe.mitre.org/ [Accessed: May 2024].

[6] National Institute of Standards and Technology (NIST), "SP 800-53 Rev. 5: Security and Privacy Controls for Information Systems and Organizations," NIST, Gaithersburg, MD, USA, Sep. 2020. [Online]. Available: https://csrc.nist.gov/publications/detail/sp/800-53/rev-5/final

[7] SANS Institute, "Web Application Security Best Practices," SANS Reading Room, 2022. [Online]. Available: https://www.sans.org/white-papers/

[8] PortSwigger, "Burp Suite Web Security Academy," 2023. [Online]. Available: https://portswigger.net/web-security/ [Accessed: May 2024].