NextGen Scientific Publication
American International Journal of Computer Science and Technology

Volume 6 Issue 1, Pg. No. 65-70, AIJCST-V6I1P107, 2024
https://doi.org/10.63282/3117-5481/AIJCST-V6I1P107

Original Article

Mitigating OWASP Top Ten Risks in Cloud-Native Healthcare and Education Platforms: A Comparative Analysis of SQL Injection and Cross-Site Scripting Defenses

***Sri Gantikota**

Senior Software Engineer, San Diego, California 92101, USA.

Abstract:

The Open Web Application Security Project Top Ten remains the most widely referenced consensus statement on web application security risks. Its 2021 revision consolidated cross-site scripting into the broader injection category, elevated broken access control to the top position, and introduced new categories for insecure design and software and data integrity failures. The categories are not abstract: they map directly to defect classes that recur across cloud-native applications regardless of domain. This paper compares mitigation patterns for two long-standing high-impact categories, SQL injection and cross-site scripting, in two regulated domains in which the author has worked: healthcare software and university research and education platforms. The two domains share the structural property that the data they handle is sensitive and regulated, but they differ in deployment topology, in user population, and in the engineering practices typical of their respective organizations. The paper documents the defenses that worked in each domain, the differences in how those defenses had to be expressed to fit the local engineering culture, and the residual risks that remained after the defenses were in place. The intent is to give practitioners a comparative reference that goes beyond restating the OWASP categories and reaches the level of operational detail at which security work actually happens.

Keywords:

OWASP Top Ten, SQL Injection, Cross-Site Scripting, XSS, Parameterized Queries, Contextual Output Encoding, Healthcare Software, Education Software, Cloud-Native, Application Security, Defense In Depth.

Article History:

Received: 22.11.2023

Revised: 24.12.2023

Accepted: 07.01.2024

Published: 18.01.2024

1. Introduction

The Open Web Application Security Project, commonly known as OWASP, publishes the OWASP Top Ten, a consensus list of the most critical security risks to web applications. The Top Ten is republished periodically as the consensus evolves. The 2021 revision is the current authoritative version at the time of this writing. The 2021 list places broken access control at position one, cryptographic failures at position two, and injection, which now includes cross-site scripting as a subcategory, at position three. Insecure design appears as a new category at position four. The remaining categories cover security misconfiguration, vulnerable and outdated components, identification and authentication failures, software and data integrity failures, security logging and monitoring failures, and server-side request forgery.

This paper focuses on two long-standing high-impact subcategories within the injection category, SQL injection and cross-site scripting, and compares mitigation patterns in two regulated domains in which the author has worked. The first domain is healthcare

software, where the data is protected health information regulated under the Health Insurance Portability and Accountability Act and the operational context is a vendor-customer relationship in which the vendor\'s engineering teams deliver software to hospital and clinic customers. The second domain is university research and education platforms, where the data is a mix of student educational records protected under the Family Educational Rights and Privacy Act, research data subject to a variety of grant and institutional review board requirements, and operational data that may carry contractual obligations. The two domains share the structural property that their data is sensitive and regulated, but they differ in deployment topology, in user population, and in the engineering practices typical of their respective organizations.

The comparison is not a ranking. Neither domain has uniformly better practices than the other. Each domain has evolved approaches that fit its own structural conditions, and each domain has residual gaps. The contribution of the paper is the documentation of the specific patterns that worked in each domain, the differences in how those patterns had to be expressed to fit local practice, and the residual risks that remained. Practitioners in either domain can use this comparison to learn from patterns developed in the other, and practitioners in adjacent regulated domains can use it as a starting point for their own analysis.

The rest of the paper is organized as follows. Section 2 covers the OWASP Top Ten 2021 with emphasis on the injection category. Section 3 compares the SQL injection defenses applied in each domain. Section 4 compares the cross-site scripting defenses. Section 5 covers the role of static and dynamic analysis tooling. Section 6 discusses defense in depth and the categories beyond injection that interact with these defenses. Section 7 covers residual risks and limitations. Section 8 concludes.

2. The OWASP Top Ten 2021 and the Injection Category

2.1. Structure of the Top Ten 2021

The 2021 revision of the Top Ten reorganized several categories that had appeared separately in prior editions. Cross-site scripting, which had its own category in 2017, was merged into the injection category. Cross-site scripting is essentially JavaScript injection into a victim\'s browser, and the root cause of mixing untrusted input with code without proper handling is the same as for the other injection forms. The reorganization clarifies that the underlying defensive pattern is consistent across the subcategories: validate, sanitize, and encode untrusted input depending on the context in which it will be used.

2.2. SQL Injection

SQL injection occurs when untrusted input is concatenated into a SQL query in a way that allows the input to alter the structure of the query. The classical example is a login form that constructs a query by string concatenation, allowing an attacker to alter the WHERE clause to bypass authentication. The defense is well established: use parameterized queries, also known as prepared statements, in which the query structure is defined in the code and the input is bound as parameters that cannot alter the structure. Object-relational mapping tools that generate parameterized queries provide the same protection when used correctly.

2.3. Cross-Site Scripting

Cross-site scripting occurs when untrusted input is rendered into a web page in a way that allows the input to execute as JavaScript in the victim\'s browser. The classical example is a comment form that renders the comment text directly into the page, allowing an attacker to include a script tag that executes when other users view the page. The defense is contextual output encoding: input is encoded according to the context in which it will appear, whether that is HTML body, HTML attribute, JavaScript, CSS, or URL. Modern web frameworks provide automatic contextual encoding for the common cases, but applications still have to take care for the cases the framework does not cover.

3. SQL Injection Defenses Compared

3.1. Healthcare Software

In the healthcare software domain, SQL injection defenses are anchored in the use of parameterized queries throughout the data access layer. The dominant frameworks in this domain produce parameterized queries by default when the developer uses the framework\'s idiomatic patterns, and the violation of the default pattern is detectable by static analysis. The defense therefore consists of two reinforcing elements: a framework that does the right thing by default and a static analysis configuration that flags departures from the default. Departures that have a legitimate reason are documented and reviewed; departures without a documented reason are treated as defects.

The dominant departure from the default pattern in this domain is the use of dynamic ORDER BY clauses in queries that support user-selected sort columns. The values of an ORDER BY clause cannot be parameterized in standard SQL because they are part of the query structure rather than data. The mitigation is to validate the user-selected sort column against a whitelist of allowed columns before constructing the query. The validation is encoded in shared helper functions so that the same logic is used everywhere, which limits the surface area on which the validation could be forgotten.

3.2. University Research and Education Platforms

In the university research and education domain, SQL injection defenses face two structural complications that are less pronounced in the healthcare domain. The first is the diversity of stacks across teams. A given university hosts applications written in nearly every common server-side language, using nearly every common framework. A defense that depends on framework-specific default behavior must be reconciled across all of these frameworks, which limits the value of any single framework\'s idiomatic patterns as the universal anchor. The second is the prevalence of analytics workloads that legitimately construct queries dynamically because the queries are themselves user-defined.

The mitigation in this domain is to invest in the language and framework-specific idioms for each stack rather than to attempt a universal pattern. Static analysis is configured per stack with the rules appropriate to that stack\'s idioms. The analytics workload case is handled by isolating the user-defined query path behind a constrained query builder that produces parameterized queries from a structured representation of the user\'s intent. The user cannot construct arbitrary SQL; they construct a query in the builder\'s vocabulary and the builder generates safe SQL on their behalf.

3.3. Comparison

The healthcare and university domains converge on the same underlying defense, parameterized queries, but they reach it through different paths because their tooling diversity is different. The healthcare domain can rely on a small number of frameworks behaving consistently. The university domain has to engineer per-stack solutions because the diversity of stacks does not support a single solution. Both domains face the same residual risk around legitimately dynamic query elements such as ORDER BY columns, and both domains address it through whitelisting against a known set of allowed values.

4. Cross-Site Scripting Defenses Compared

4.1. Healthcare Software

In the healthcare software domain, cross-site scripting defenses are anchored in template engines that perform contextual output encoding by default. The same two-element pattern from SQL injection applies: a framework that does the right thing by default and a static analysis configuration that flags departures. Departures in this domain are most often associated with rich-text fields, such as the free-text notes a clinician writes about a patient encounter. Rich-text fields are intentionally rendered with HTML markup, which means that an aggressive contextual encoding would strip the formatting the user expects to see.

The mitigation for rich-text fields is to use an HTML sanitizer that allows a documented subset of HTML elements and attributes while removing anything that could execute as JavaScript. The sanitizer is configured with a strict allowlist tuned to the formatting capabilities the clinical workflow actually needs. New attributes or new elements are not added to the allowlist without a security review. The sanitizer is applied at render time rather than at storage time so that a sanitizer update can apply to historical content without requiring a database rewrite.

4.2. University Research and Education Platforms

In the university research and education domain, cross-site scripting defenses face a complication that healthcare typically does not: the wide use of academic content management systems and learning management systems in which instructors and students are expected to provide rich content including embedded media, mathematical notation rendered through libraries such as MathJax, and code snippets with syntax highlighting. The set of HTML constructs that the platform must render is therefore broader than in the healthcare domain, and the sanitizer\'s allowlist must accommodate this breadth without opening cross-site scripting paths.

The mitigation in this domain follows the same architectural pattern as healthcare, an allowlist-based sanitizer applied at render time, but the allowlist is broader and is reviewed more frequently as instructional needs evolve. Additional defenses include Content Security Policy headers that limit the sources from which scripts can be loaded and that disable inline scripts where possible.

Content Security Policy is also useful in the healthcare domain but is more prominent in the education domain because the diversity of legitimate content forms in the education domain makes a single allowlist-based sanitizer less able to carry the full defensive weight.

4.3. Comparison

The healthcare and university domains again converge on the same underlying defenses, contextual output encoding by default plus allowlist-based sanitization for rich content, but the university domain leans more heavily on Content Security Policy as a complementary control because the diversity of legitimate content forms is broader. Both domains apply sanitization at render time rather than at storage time so that sanitizer updates apply to historical content. Both domains accept that the rich-text path is the residual risk surface where the defense is least automatic.

5. Static and Dynamic Analysis Tooling

5.1. SAST Configuration

Static Application Security Testing tooling such as SonarQube provides rules that detect both SQL injection and cross-site scripting patterns. The default rule set is conservative and produces both true positives and false positives. The cost of tuning the rule set per project is real but bounded. In both the healthcare and university domains, the tuning effort pays back quickly because the alternative is either developers ignoring the findings as noise or the security team triaging the same false positives repeatedly. Tuning involves enabling stack-specific rules, disabling rules that produce systematic false positives in the codebase, and configuring sources and sinks for taint analysis.

5.2. DAST Coverage

Dynamic Application Security Testing tooling complements SAST by exercising the application from the outside. DAST is particularly useful for cross-site scripting because the encoding behavior of a template engine can sometimes be defeated by a content path the static analysis does not see, and DAST will exercise the path that the user-facing rendering actually takes. DAST is also useful for SQL injection in legacy codebases where the data access layer is heterogeneous enough that static analysis cannot reliably enumerate all the query sites.

5.3. Penetration Testing

Penetration testing covers what SAST and DAST miss. The most common categories of finding from penetration testing in these domains are business-logic flaws that depend on the application\'s workflow rather than on patterns visible in the code or the response stream. Business-logic flaws are not in the injection category, but the engagement of a penetration tester is the practical mechanism by which an organization confirms that its injection defenses hold under adversarial probing. Penetration testing in healthcare is often performed by a third-party firm under a signed engagement. Penetration testing in the university domain may be performed by a central security team\'s internal penetration testers or by a third party depending on the application.

6. Defense in Depth beyond Injection

6.1. Broken Access Control

Broken access control is the top category in the 2021 Top Ten and interacts with injection defenses in a specific way: an authenticated user who can exploit a SQL injection or cross-site scripting flaw can often escalate their access through that flaw. Access control defenses that limit what an authenticated user can see and do reduce the impact of an injection that nonetheless slips through. Both domains apply role-based and attribute-based access control patterns; the patterns are not different between the two domains in any significant way.

6.2. Cryptographic Failures

Cryptographic failures, the second category, intersect with injection defenses around the storage of credentials. Passwords stored using weak hashing such as unsalted MD5 are vulnerable to cracking if exfiltrated through an injection. The standard mitigation, password hashing with a modern algorithm such as Argon2 or bcrypt with a strong cost parameter, is applied in both domains. The healthcare domain has additional encryption-at-rest requirements that come from the HIPAA Security Rule; the university domain has comparable requirements where research data is subject to data use agreements that mandate encryption.

6.3. Security Logging and Monitoring Failures

Security logging and monitoring failures, the ninth category, matter for injection defenses because the value of catching an attempted exploitation in the logs depends on having the logs in the first place. Both domains log authentication events, authorization failures, and rejected inputs at the boundary. The logs are routed to a central system that can correlate events across applications. The healthcare domain has additional auditing requirements that the HIPAA Security Rule mandates; the university domain has additional requirements where regulated data is involved.

6.4. Insecure Design

Insecure design, the fourth category, captures the case where the application\'s structure makes a class of vulnerabilities easy to introduce or hard to detect. Design reviews early in the development lifecycle, threat modeling for new features, and the use of secure design patterns are the standard mitigations. Both domains practice these to varying degrees. The healthcare domain tends to have more formal design review because of regulatory expectation; the university domain tends to have more informal review tied to academic project supervision.

7. Residual Risks and Limitations

7.1. The Legacy Long Tail

Both domains carry legacy applications that predate the modern defensive patterns. These applications are too valuable to retire and too entangled with their data to rewrite. The defensive approach for legacy applications relies more heavily on perimeter controls such as web application firewalls and on close monitoring than on the in-application controls that newer applications can use. The legacy long tail is the residual risk surface that defies clean architectural treatment in either domain.

7.2. Third-Party Components

Third-party components in either domain may carry their own injection vulnerabilities that the application\'s defenses do not protect against. Software Composition Analysis tooling detects known vulnerabilities in declared dependencies but does not protect against vulnerabilities the upstream has not yet disclosed. Both domains accept this residual risk and mitigate it through prompt upgrade discipline once vulnerabilities are disclosed.

7.3. Developer Education

Developer education is a continuing investment in both domains. New developers join, frameworks evolve, and the specific patterns that constitute safe code change over time. The investment includes onboarding material that covers the local defensive patterns, periodic refresher content on the OWASP categories that the codebase most often touches, and pair programming sessions that propagate the patterns through demonstration.

8. Conclusion

SQL injection and cross-site scripting remain among the most consequential web application security risks despite being well-understood for many years. The defensive patterns are also well-understood: parameterized queries for SQL injection, contextual output encoding for cross-site scripting, with allowlist-based sanitization for the rich-content cases that simple encoding cannot handle. The comparative analysis in this paper between healthcare software and university research and education platforms shows that the underlying patterns are the same across the two domains, but that the patterns must be expressed differently to fit the engineering practices of each domain. Healthcare can lean on a small number of frameworks behaving consistently by default. Universities have to engineer per-stack solutions because the diversity of stacks does not support a single solution. Both domains converge on the same architectural approach and on the same residual risks. Practitioners in either domain can learn from patterns developed in the other, and practitioners in adjacent regulated domains can use this comparison as a starting point for their own work.

Acknowledgments

This work draws on engineering practice in healthcare software development at IBM and Merge Healthcare and on security and cross-functional collaboration at the University of California, San Diego. The author thanks colleagues across both organizations whose practice informed this analysis.

Conflicts of Interest

The author declares that there is no conflict of interest concerning the publishing of this paper.

References

[1] Open Web Application Security Project. OWASP Top Ten Web Application Security Risks, 2021 edition. https://owasp.org/Top10/2021/ | https://scholar.google.com/scholar?hl=en&q=OWASP Top Ten Web Application Security Risks, 2021 edition

[2] Open Web Application Security Project. OWASP Top Ten, A03 Injection, 2021 edition. https://scholar.google.com/scholar?hl=en&q=OWASP Top Ten, A03 Injection, 2021 edition.

[3] Open Web Application Security Project. OWASP Cross-Site Scripting Prevention Cheat Sheet. https://scholar.google.com/scholar?hl=en&q=OWASP Cross-Site Scripting Prevention Cheat Sheet

[4] Open Web Application Security Project. OWASP SQL Injection Prevention Cheat Sheet. https://scholar.google.com/scholar?hl=en&q=OWASP SQL Injection Prevention Cheat Sheet

[5] Open Web Application Security Project. OWASP Application Security Verification Standard, Version 4.0.3. https://scholar.google.com/scholar?hl=en&q=OWASP Application Security Verification Standard, Version 4.0.3.

[6] Open Web Application Security Project. OWASP Secure Coding Practices Quick Reference Guide. https://scholar.google.com/scholar?hl=en&q=OWASP Secure Coding Practices Quick Reference Guide

[7] Common Weakness Enumeration. CWE-89: Improper Neutralization of Special Elements used in an SQL Command, MITRE Corporation. https://scholar.google.com/scholar?hl=en&q=CWE-89: Improper Neutralization of Special Elements used in an SQL Command, MITRE Corporation

[8] Common Weakness Enumeration. CWE-79: Improper Neutralization of Input During Web Page Generation, MITRE Corporation. https://scholar.google.com/scholar?hl=en&q=CWE-79: Improper Neutralization of Input During Web Page Generation, MITRE Corporation

[9] Common Weakness Enumeration. CWE Top 25 Most Dangerous Software Weaknesses. https://scholar.google.com/scholar?hl=en&q=CWE Top 25 Most Dangerous Software Weaknesses

[10] World Wide Web Consortium. Content Security Policy Level 3, W3C Working Draft. https://scholar.google.com/scholar?hl=en&q=Content Security Policy Level 3, W3C Working Draft

[11] SonarSource. SonarQube static analysis platform documentation. https://scholar.google.com/scholar?hl=en&q=SonarQube static analysis platform documentation

[12] International Business Machines Corporation. IBM Security AppScan documentation. https://scholar.google.com/scholar?hl=en&q=IBM Security AppScan documentation

[13] United States Department of Health and Human Services. Health Insurance Portability and Accountability Act Security Rule, 45 CFR Part 164 Subpart C. https://scholar.google.com/scholar?hl=en&q=Health Insurance Portability and Accountability Act Security Rule, 45 CFR Part 164 Subpart C

[14] United States Department of Education. Family Educational Rights and Privacy Act, 20 U.S.C. 1232g. https://scholar.google.com/scholar?hl=en&q=Family Educational Rights and Privacy Act, 20 U.S.C

[15] National Institute of Standards and Technology. Secure Software Development Framework, NIST Special Publication 800-218. https://scholar.google.com/scholar?hl=en&q=Secure Software Development Framework, NIST Special Publication 800-218

[16] National Institute of Standards and Technology. Security and Privacy Controls for Information Systems and Organizations, NIST Special Publication 800-53 Revision 5. https://scholar.google.com/scholar?hl=en&q=Security and Privacy Controls for Information Systems and Organizations, NIST Special Publication 800-53 Revision 5

[17] Provos, N. and Mazieres, D. A Future-Adaptable Password Scheme. USENIX, 1999. https://scholar.google.com/scholar?hl=en&q=and Mazieres, D

[18] Biryukov, A., Dinu, D., and Khovratovich, D. Argon2: New Generation of Memory-Hard Functions for Password Hashing and Other Applications. IEEE EuroS and P, 2016. https://scholar.google.com/scholar?hl=en&q=Argon2: New Generation of Memory-Hard Functions for Password Hashing and Other Applications

[19] Howard, M. and Lipner, S. The Security Development Lifecycle. Microsoft Press, 2006. https://scholar.google.com/scholar?hl=en&q=and Lipner, S