An Empirical Study on Security Misconfigurations in TypeScript Backends in the Open Source Academic Projects

Leandro Braga
Federal University of Itajubá - UNIFEI
Itajubá, Brazil
lbbraga@proton.me

Phyllipe Francisco
Federal University of Itajubá - UNIFEI
Itajubá, Brazil
phyllipe@unifei.edu.br

Bruno Batista
Federal University of Itajubá - UNIFEI
Itajubá, Brazil
brunoguazzelli@unifei.edu.br

Abstract

Security misconfigurations represent a persistent threat to modern web systems. Considering the rise of TypeScript-based backend systems in academic applications, there is still a lack of guidelines ensuring that these security measures are taken into account, even though the theory is usually well-documented in the frameworks' official documentation. This work proposes an empirical study of open-source academic projects to identify vulnerabilities in TypeScript systems. The methodology combines mining software repositories (MSR) and static code analysis. The tool VulnSniffer was developed to automate data extraction from the selected repositories. A total of 164 repositories linked to academic projects were selected for analysis. To identify the vulnerabilities, this work conducted three studies. First, VulnSniffer automatically identifies vulnerabilities; then, the tool allows the author to manually audit each detected vulnerability to determine the automated tool's accuracy; finally, a questionnaire was administered to a cybersecurity expert to analyze the quality of both the automated analysis and the manual audit. Our results show that 72 repositories exhibited at least one vulnerability identified automatically by the tool. After a manual audit, 45 repositories were confirmed to have at least one vulnerability

Keywords

Mining Software Repository, Security Misconfigurations, Source COde Analysis, TypeScript,Web Vulnerabilities

1 Introduction

Web systems have become an essential part of modern society. They are dynamic applications that enable everything from global e-commerce to social interactions. These platforms operate on a distributed architecture, in which business logic, data processing, and communication with databases occur on the *backend* [2]. This component manages a large volume of sensitive information, including access credentials, personal data, and financial transactions, making its security a fundamental requirement for the trustworthiness and operation of digital services [10].

The very complexity of modern systems increases the number of vulnerable points that can be exploited by *hackers* to carry out cyberattacks. As a result, software security has become a field engaged in a continuous battle against evolving threats. The *Open Web Application Security Project (OWASP)* maintains a global consensus on the most critical risks through the *OWASP Top 10* report [14]. Although this report has traditionally focused on existing vulnerabilities, recent trends indicate a continuous evolution in which security misconfiguration remains one of the most prevalent and dangerous threats. These weaknesses often stem from implementation errors, insecure default configurations, verbose error messages, or outdated software components.

The importance of addressing this challenge has been recognized in the academic literature [1]. In particular, software systems developed in academic environments, although intended primarily for educational purposes, often reach the complexity of real-world production applications without receiving equivalent attention to information security. Therefore, academic repositories are an interesting subject of study for assessing the security maturity achieved in the education of future software developers.

The main objective of this research is to investigate and quantify the prevalence of common security misconfigurations in open-source TypeScript *backend* applications used in Brazilian academic projects. Through the Mining Software Repository (MSR) and Source Code Analysis, this work seeks to produce an empirical map of the most selected security issues.

To support the data collection process, the open-source software tool `VulnSniffer Tool`[1] was developed. It was designed to perform static analysis of TypeScript source code (files with the "`.ts`" extension), identify predefined security configuration patterns, and generate a quantitative report containing the collected metrics.

The tool selected a sample of 164 open-source software projects hosted on GitHub. One of the selection criteria was their **association with academic institutions**, as previous studies indicate that academic repositories may present specific, often underestimated security challenges [5]. This criterion was adopted to generate evidence that may highlight the importance of strengthening secure software development education in universities.

As a result, among the 164 selected projects, 72 contained at least one vulnerability automatically identified by the tool. Of these, 45 projects were confirmed through manual auditing to contain at least one genuine vulnerability, corresponding to a compromise rate of 62.5% of the analyzed sample.

2 Background

Information security has become essential to managing daily activities and promoting social well-being and healthcare in today's society. During the COVID-19 pandemic, reliance on technology increased significantly for various purposes, including access to healthcare services, maintaining social connections, and meeting basic daily needs. The pandemic forced many everyday activities to move online [4]. In this context, Information Security is established as the discipline dedicated to protecting these assets, encompassing not only technology but also the processes and people involved in safeguarding data against threats [11].

[1] https://github.com/leandrobalta/VulnSniffer

According to International Organization for Standardization [7], the conceptual framework of Information Security is traditionally based on three fundamental pillars: confidentiality, integrity, and availability, whose definitions have been consolidated in international management standards. These pillars are defined as follows:

- **Confidentiality:** ensures that information is accessible only to duly authorized individuals, entities, or processes, protecting data against unauthorized disclosure.
- **Integrity:** ensures the accuracy and completeness of information, guaranteeing that it has not been altered or corrupted in an unauthorized manner.
- **Availability:** ensures that authorized users have access to information and its associated assets whenever needed, thereby maintaining service continuity.

To guide developers and security professionals regarding risks in web applications, the *Open Web Application Security Project (OWASP)* maintains the *OWASP Top 10* report, which serves as an awareness document for the software industry [9].

A security misconfiguration is defined as an error in system configuration or security settings that leaves systems vulnerable to attacks. Such errors provide attackers (*hackers*) with an easy entry point to exploit. Security misconfigurations manifest across various domains, including network configurations (such as improperly configured *firewall* rules), application settings (such as default passwords), and cloud computing environments.

The complexity of modern computing environments means that security misconfigurations continue to represent a substantial risk, leading to unintended data exposure, security breaches, and financial losses [6]. Configuration flaws can manifest in several ways, often resulting from the omission of basic security settings [13].

According to Dietrich et al. [3], studies examining system operators' perspectives indicate that configuration failures are often driven by a combination of human, organizational, and systemic factors rather than by individual negligence alone. Time pressure, the absence of review processes, excessive tool complexity, and organizational cultures that fail to prioritize security are identified as the primary catalysts for these errors.

Effectively mitigating security misconfigurations requires a "defense-in-depth" strategy that combines technology, processes, and organizational culture. In addition to regular security audits and the automation of *hardening* processes, the adoption of Static Application Security Testing (SAST) tools has become a well-established approach to inspecting source code for vulnerability patterns, enabling the detection of security flaws before deployment [8].

3 Research Design

To achieve the objectives of this study, the following steps were carried out:

(1) **Develop the VulnSniffer Tool.**
(2) **Define the Repository Selection Criteria.**
(3) **Identify the Vulnerabilities of Interest.**
(4) **Conduct an Exploratory Data Analysis.**
(5) **Apply a Questionnaire to an Expert.**

3.1 Developed Tool

To support the data collection process for this research, the VulnSniffer was developed. It can read and interpret TypeScript files (".ts") and extract information from the source code. After the data collection process, a "csv" report is generated. The tool provides three main functionalities, described as follows:

(1) Select repositories according to the criteria described in Section 3.2;
(2) Clone the selected repositories locally and traverse each ".ts" file to identify information related to *backend* vulnerabilities. The tool also classifies the detected vulnerabilities.
(3) The tool allows the user to manually classify each vulnerability identified by the automated analysis (previous step (2)) through an interactive command-line interface. The interface displays the project title, project description (when available), and the source code snippet that potentially contains a security misconfiguration. Based on this information, the user may assign one of the following classifications:
   - **Correct:** the vulnerability identified by the tool is indeed a genuine vulnerability.
   - **Incorrect:** the vulnerability identified by the tool is **not** a genuine vulnerability.
   - **Not Applicable:** regardless of the analysis outcome, the repository/project is outside the scope of this study.

3.2 Repository Selection Criteria

The data used in this study consists of open-source repositories hosted on GitHub. A total of 164 repositories were selected using the VulnSniffer, which applied the following selection criteria:

- Use the TypeScript programming language;
- Have at least one GitHub star;
- Include at least one of the *backend* libraries/*frameworks* listed in Table 1 in the *package.json* file.

Table 1: Analyzed Frameworks and Libraries

Framework/Library	Package.json String
Express	`express`
NestJS	`nestjs, @nestjs/core`
Fastify	`fastify`
Koa	`koa`
TypeORM	`typeorm`
Prisma	`prisma`
Mongoose	`mongoose`
Sequelize	`sequelize`
Passport	`passport`
Socket.IO	`socket.io-client`
Symfony	`symfony/webpack-encore`
Hono	`hono`

The selection of these libraries/frameworks was based on their maturity and widespread adoption in TypeScript *backend* development. The most widely adopted libraries and *frameworks* within the community were selected, including Express, NestJS, and Fastify, as

well as well-established and widely used data persistence solutions such as TypeORM, Prisma, and Mongoose [12].

3.3 Vulnerabilities of Interest

This section presents the configuration and implementation vulnerabilities considered in this study. The selection of the security misconfigurations analyzed was guided by both the technical feasibility of automated detection and their practical relevance. Accordingly, four categories of security flaws were selected, all aligned with category A02:2025 (*Security Misconfiguration*) of the OWASP Top 10 [9].

The primary reason for selecting this specific set of vulnerabilities is that they exhibit distinctive structural patterns and textual content within TypeScript source code. This characteristic enables reliable identification and automated extraction via static analysis using the Abstract Syntax Tree (AST). Furthermore, these vulnerabilities reflect common, critical configuration decisions in popular *frameworks* and tools.

The vulnerabilities are described below.

- **Hardcoded Secret:** Confidential variables written directly into the source code represent one of the most critical configuration and development flaws in *backend* applications. This includes database passwords, access credentials, certificates, and any other sensitive information stored in plain text within configuration files, environment variables, or application constants.
- **Database Auto Sync:** Although useful during the development phase, automatic database synchronization is considered risky in production environments. In Object-Relational Mapping (ORM) libraries such as TypeORM, automatic synchronization attempts to update the database schema based on the entities defined in the source code. This approach reduces control over schema changes and increases the likelihood of unintended modifications to the database structure.
- **Insecure CORS:** Cross-Origin Resource Sharing (CORS) is a browser security mechanism that controls whether a web page may access resources hosted on another domain. An insecure CORS configuration occurs when a server accepts requests from untrusted origins or grants overly permissive access. A misconfigured CORS policy may turn the user's browser into an unintended intermediary for requests it should not accept.
- **Mass Assignment (NestJS):** Mass assignment occurs when an application forwards all fields received in a request directly to an entity, model, or repository without filtering which attributes are allowed to be modified. In NestJS, this issue commonly arises when developers use the request body directly in database operations without first mapping it to a well-defined Data Transfer Object (DTO) and without restricting the accepted fields. The risk associated with this vulnerability lies in allowing clients (users) to modify structural parameters that should be managed exclusively by the business logic implemented on the *backend* server.

3.4 Exploratory Data Analysis

After the *VulnSniffer* tool completed Stages 1 (repo cloning) and 2 (automatic source code analysis), it extracted potential vulnerabilities from the analyzed projects. Subsequently, the third stage, referred to as the auditing phase, provides an interactive command-line interface through which the user can inspect the title, description, and source code snippet associated with each detected issue before assigning its classification.

The manual auditing process was integrated into the exploratory data analysis as a validation step to confirm the academic nature of the selected repositories, verify their Brazilian origin, and determine whether the identified source code fragment genuinely constitutes a security misconfiguration.

Consequently, in addition to quantifying the identified security flaws, the analysis provided a contextualized overview of the Brazilian academic landscape regarding the maturity of security practices and the adoption of secure development practices in TypeScript *backend* systems. Based on the collected data, this study sought to answer the following research questions:

(i) Are configuration vulnerabilities (*Security Misconfigurations*) significantly prevalent in Brazilian open-source academic software projects?
(ii) Which types of vulnerabilities exhibit the highest recurrence and prevalence across these repositories?

3.5 Expert Validation Questionnaire

To validate the manual analysis criteria adopted in this study, mitigate potential bias introduced by the author during the auditing process, and assess the practical accuracy of the security misconfiguration classification, an external validation stage was conducted through a questionnaire, in which an expert evaluated a sample of the audited vulnerabilities.

The invited expert has extensive professional experience as both a software engineer and an application security engineer. In addition, they have an extensive record of active contributions to open-source projects, with a particular focus on identifying and responsibly reporting structural vulnerabilities.

The sample selected for the validation questionnaire was drawn using a stratified random sampling strategy, corresponding to approximately 10% of the repository population. This procedure yielded a sample of 15 distinct projects, as shown in Table 2, that were submitted for independent review. This approach ensured that both the positive and negative accuracy of the auditing methodology could be evaluated by the expert with statistical significance.

The sampled instances were divided into two groups:

(1) **Correct Vulnerabilities:** source code fragments in which the automated tool identified security misconfigurations, and the author's manual audit confirmed the finding as a genuine and exploitable vulnerability. In Table 2 it has status "OK".
(2) **Incorrect Vulnerabilities:** source code fragments in which the automated tool identified potential security misconfigurations; however, the manual audit by the author determined that the reported issue was a false positive due to the application's context. In Table 2 it has status "NOK".

Table 2: Selected Sample for Expert Evaluation

Repo	File	Type	Status
https://github.com/anniasebold/subscription-api	src/main.ts	NESTJS_MASS_ASSIGNMENT	OK
https://github.com/Hyagobsantos/Api-LojaStarWars	src/auth/jwt.constants.ts	HARDCODED_SECRET	OK
https://github.com/joaowinkelmann/finalflow-api	src/main.ts	NESTJS_MASS_ASSIGNMENT	OK
https://github.com/ecirilo/ufsj-tecweb-20242-backend	src/app/repositories/datasource.provider.ts	DB_AUTO_SYNC	OK
https://github.com/LucasSeraggi/TFG-Backend	src/app.ts	INSECURE_CORS	OK
https://github.com/disouzam/atividades-arq-aplicacoes-nodejs-puc-minas-2025	apps/tasks/src/infrastructure/infrastructure.module.ts	DB_AUTO_SYNC	OK
https://github.com/GuiDev115/nest-api-article	src/config/database.config.ts	DB_AUTO_SYNC	NOK
https://github.com/KayoRonald/euajudo-back-end	src/shared/http/server.ts	INSECURE_CORS	NOK
https://github.com/luissimas/pooa-grupos-academicos	src/infra/repositories/event/memoryEventRepository.ts	HARDCODED_SECRET	NOK
https://github.com/tpiva/ts-api-me	server/config/env/development.env.ts	HARDCODED_SECRET	NOK
https://github.com/diname/tech-challenge-fiap	src/domain/services/auth/auth.service.impl.ts	HARDCODED_SECRET	NOK
https://github.com/brunokobi/tccbackend	src/server.ts	INSECURE_CORS	OK
https://github.com/Jao-Rocha/onebitflix	src/adminjs/authentication.ts	HARDCODED_SECRET	NOK
https://github.com/JhonyDomingos/backend-just-connect	generated/prisma/internal/prismaNamespace.ts	HARDCODED_SECRET	NOK
https://github.com/NessKawl/API_driveparts	src/auth/strategies/local.strategy.ts	HARDCODED_SECRET	NOK

The **Questionnaire**[2] and the **Spreadsheet**[3] containing the responses are publicly available online.

4 Results and Discussion

The initial mining process yielding a sample of 164 projects. Figure 1 presents the frequency distribution of the identified *backend* technologies. This metric reflects the cumulative use of these technologies, since a single repository may employ multiple frameworks and libraries simultaneously.

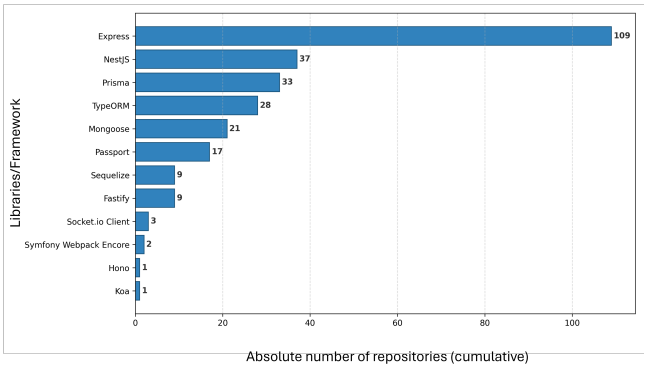

Figure 1: Frequency Distribution of Backend Frameworks and Technologies in the Selected Sample.

From the mined projects, the tool pointed that a total of 72 unique repositories had at least 1 vulnerability. These projects were submitted to a manual audit by the author. The results revealed that 45 projects contained at least one confirmed security vulnerability, corresponding to a compromise rate of 62.5% of the analyzed sample.

Overall, the analysis confirmed 62 instances of *Security Misconfigurations*, distributed across the four evaluated categories. Table 3 summarizes the absolute and relative distribution of these occurrences, as well as the accuracy metrics achieved by the `VulnSniffer` tool.

Table 3: Distribution of Confirmed Vulnerabilities

Vulnerability	Total	Confirmed	Affected Projects	Tool Accuracy(%)
Insecure CORS	44	29	29	65,9
Mass Assignment (NestJS)	19	13	10	68,4
Database Auto Sync	14	12	11	85,7
Hardcoded Secret	82	8	6	9,7

The results indicate that vulnerabilities related to Cross-Origin Resource Sharing (CORS) and the absence of restrictive validation policies in persistence and data access layers account for the largest proportion of security flaws found in software developed by students.

4.1 Identified Vulnerabilities

Insecure CORS was the most prevalent vulnerability identified in this study, with 29 validated instances distributed across 29 distinct repositories. This indicates that approximately 40.0% of all TypeScript *backend* projects included in the sample employed overly permissive access control policies.

From a qualitative perspective, the analysis revealed a recurring pattern of allowing requests from all origins and enabling CORS without any filtering parameters. Within the academic context, this behavior can be explained by practical and time-constrained development decisions. During the early stages of development, students frequently encounter cross-origin request blocking errors when integrating a client application (*frontend* running locally) with the API server (*backend*).

NestJS Mass Assignment presented a total of 13 instances identified and manually validated across 10 repositories based on the NestJS *framework*. This vulnerability was primarily caused by the absence or misconfiguration of the global input validation component (`ValidationPipe`).

In practice, this vulnerability allows a malicious user to submit additional request fields that should not be modifiable, such as

[2]https://forms.gle/jDSSfswoWPyHnhLi8
[3]https://docs.google.com/spreadsheets/d/1WFNTCBqe68PVmpnNJ38E9VQFnu_Zt17ngxT1DAeTeqI/edit?usp=sharing

elevating their own privileges to administrator or modifying an account balance.

Database Auto Sync) was identified in 12 instances distributed across 11 different academic projects. This scenario involves enabling the synchronization property in Object-Relational Mapping (ORM) configuration files, such as those used by TypeORM or Sequelize.

Although automatic synchronization is a highly productive feature during the early stages of local development, leaving this option enabled in production environments poses severe risks to data integrity. Minor changes to the source code, renaming an entity column, or even a simple application restart may trigger automatic schema modification (`ALTER`) or table deletion (`DROP`) commands, potentially resulting in irreversible data loss.

Hardcoded Secrets was initially identified by 82 potential alerts. However, the manual auditing phase revealed one of the most significant qualitative findings of this study: only 8 instances were confirmed as genuine hardcoded credentials exposed in plain text. The remaining 74 alerts were classified either as false positives or as non-applicable cases.

This occurred because the syntactic detection rule implemented in the `VulnSniffer` identified literal text (*string*) values associated with object properties and variables containing generic names such as `secret`, `passwordField`, or `token`. In most of the audited cases, these variables represented internal authentication strategy settings or test variables rather than production infrastructure credentials that posed an actual security risk.

4.2 Algorithm Accuracy and the Impact of False Positives

To assess the effectiveness of the data obtained through the automated analysis, the author conducted a manual validation. Figure 2 illustrates this discrepancy across the evaluated vulnerability categories.

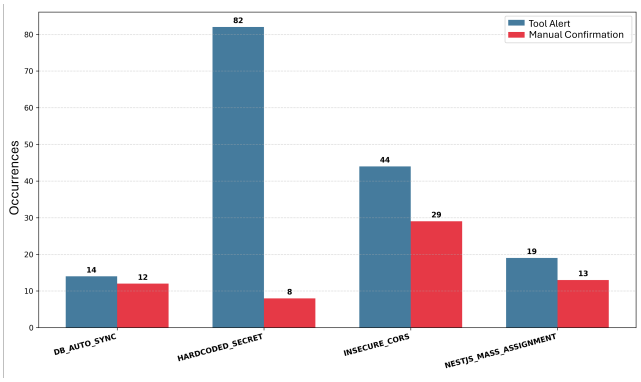

Figure 2: Accuracy of AST-Based Analysis Compared with Manual Audit Validation.

As shown in Figure 2, the `HARDCODED_SECRET` category represents the greatest challenge for purely syntactic analysis tools. In contrast, more structurally constrained categories, such as `DB_AUTO_SYNC`, achieved an accuracy rate of 85.7%, largely because their semantic patterns are easier to identify. In most cases, these vulnerabilities are characterized by explicit source code statements such as `synchronize: true`.

4.3 Expert Evaluation Results

To further establish the reliability of the auditing procedure proposed in this study, a subset of the results obtained during the auditing stage by the author was independently evaluated by an external *Application Security* expert. The evaluation sample consisted of 15 repositories, including both cases previously validated as genuine vulnerabilities (10 instances) and cases classified during the manual audit as false positives (five instances).

Table 4 summarizes the results of the questionnaire completed by the expert, as described in Section 3.5. In the Expert Assessment column, the value "OK" means the expert agreed with vulnerability pointed automatically by the tool.

Table 4: Results of the Expert Evaluation

Repository	Expert Assessment	Author's Assessment
subscription-api	OK	OK
Api-LojaStarWars	OK	OK
finalflow-api	OK	OK
ufsj-tecweb-20242-backend	OK	OK
TFG-Backend	OK	OK
aplicacoes-nodejs-puc-minas-2025	NOK	OK
nest-api-article	NOK	NOK
euajudo-back-end	OK	NOK
pooa-grupos-academicos	OK	NOK
ts-api-me	NOK	NOK
tech-challenge-fiap	OK	NOK
tccbackend	OK	OK
onebitflix	OK	NOK
backend-just-connect	NOK	NOK
API_driveparts	NOK	NOK

As seen in Table 4, there was not an 100% agreement between the author and the expert analysis The **agreement rate** was 66.67% (10 out of the 15 evaluated instances) with the author's manual audit. When these results are compared with the automated detections produced by the `VulnSniffer`, different levels of agreement emerge. The author agreed with the automated classifications in 46.6% of the cases, whereas the expert agreed with the automated detections in 66.67% of the cases. Furthermore, when agreement between the expert and the author is analyzed by vulnerability category, distinct patterns emerge across the different vulnerability types, as illustrated in Figure 3.

These discrepancies highlight an important methodological finding. The initial manual audit adopted by the author was **more conservative**, evaluating the identified security misconfigurations in the broader context of academic software projects. In contrast, the expert's assessment followed a more technical perspective aligned with industry security practices, in which determining whether a security misconfiguration exists depends primarily on the overall project context rather than on the isolated source code fragment. This difference in perspective strengthens the study by demonstrating that interpretations of security misconfigurations can vary between an initial auditor and an experienced application security professional, emphasizing the importance of contextual analysis in vulnerability assessment.

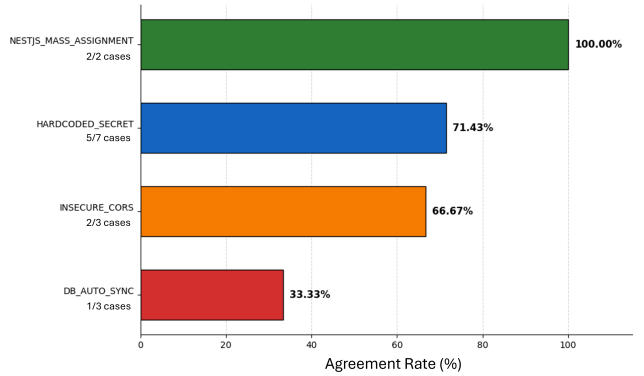

Figure 3: Agreement Rate with the Expert by Vulnerability Category

4.4 Discussion of the Research Questions

Consolidating both quantitative and qualitative findings enables answers to the research questions guiding this study.

Research Question (i): *Are configuration vulnerabilities (Security Misconfigurations) significantly prevalent in Brazilian open-source academic software projects?*

Yes. The presence of security misconfigurations is significant. The finding that 62.5% of the audited academic projects contained at least one confirmed security misconfiguration supports the hypothesis that information security is frequently treated as a secondary concern during undergraduate software development education. The results suggest a clear prioritization of fulfilling the software's functional requirements (i.e., making the application work) over non-functional security requirements.

Research Question (ii): *Which types of vulnerabilities exhibit the highest recurrence and prevalence across these repositories?*

As demonstrated by the collected data, **Insecure CORS** was the most prevalent vulnerability, affecting 29 distinct projects (approximately 40.0% of the entire sample). It was followed by **NestJS Mass Assignment** (13 confirmed instances) and **Database Auto Sync** (12 confirmed instances). This distribution indicates that the most common vulnerabilities are directly associated with the absence of secure configuration practices, often adopted to simplify application integration and accelerate development. In other words, the findings suggest that achieving a functional implementation was frequently prioritized over establishing secure software configurations.

5 Conclusion

This work investigated the extent to which academic TypeScript *backend* projects adhered to best practices for security configurations. To reach this goal, the proposed study adopted Mining Software Repositories (MSR) as its primary methodological approach, combined with static source code analysis performed using the VulnSniffer tool, developed specifically for this research.

The repository mining process yielded 72 repositories, which the author manually audited. The findings were straightforward: 45 of these projects, corresponding to 62.5% of the analyzed sample, contained at least one confirmed security misconfiguration. In total, 62 legitimate instances of security misconfigurations were identified across the four vulnerability categories considered in this study.

To validate both the automated detection results and the manual auditing process, an external application security expert evaluated a sample of the analyzed data. This validation resulted in 66.67% agreement with the author's manual audit findings.

Among the identified vulnerabilities, *Insecure CORS* was the most prevalent. This result is particularly noteworthy because the vulnerability can be prevented without advanced technical expertise. In most cases, it stemmed from a development decision made during implementation to resolve connectivity issues between the client and the server.

Regarding threats to validity, the repository selection criteria constitute an important limitation, since the set of projects available on GitHub changes continuously over time. Furthermore, the scope of this study was limited to four categories of security misconfigurations. Finally, the manual auditing was conducted by one author and compared with one application security expert. Ideally, three experts (at least) should be interviewed and invited to assess these results and the tools' automatic extraction.

Several directions for future work emerge from this study. The first is to expand the set of analyzed vulnerabilities by incorporating additional OWASP Top 10 categories that can be identified through static analysis. Another promising direction is to perform a longitudinal analysis to investigate whether the identified vulnerabilities were corrected throughout the software lifecycle or remained unchanged since the earliest *commits*. Finally, we are working on inviting more experts to contribute to the tools' automatic extraction and assessment processes.

Artifact Availability

- VulnSniffer Tool: https://github.com/leandrobalta/VulnSniffer
- Expert Answers: https://docs.google.com/spreadsheets/d/1WFNTCBqe68PVmpnNJ38E9VQFnu_Zt17ngxT1DAeTeqI/edit?usp=sharing
- Questionnaire: https://forms.gle/jDSSfswoWPyHnhLi8

Acknowledgements

In this paper, the artificial intelligence tools Gemini 3.5 Flash (Google) and Claude 4.5 Haiku (Anthropic) were used to generate graph plotting algorithms, correct LaTeX syntax errors, perform grammatical revisions, and draft the textual descriptions of the repositories used in the questionnaire administered to the domain expert. The authors declare that these tools were used solely as support instruments.

References

[1] Sultan S. Alqahtani, Ellis E. Eghan, and Juergen Rilling. 2016. Tracing known security vulnerabilities in software repositories: A Semantic Web enabled modeling approach. *Science of Computer Programming* 121 (2016), 153–175.

[2] Amazon Web Services. 2025. What is a Web Application? Disponível em: https://aws.amazon.com/what-is/web-application/.

[3] Constanze Dietrich, Katharina Krombholz, Kevin Borgolte, and Tobias Fiebig. 2018. Investigating System Operators' Perspective on Security Misconfigurations. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18)*. 18. doi:10.1145/3243734.3243794

[4] Sarah J. Dow-Fleisner, Cherisse L. Seaton, Eric Li, Katrina Plamondon, Nelly Oelke, Donna Kurtz, Charlotte Jones, Leanne M. Currie, Barb Pesut, Khalad Hasan, and Kathy L. Rush. 2022. Internet access is a necessity: a latent class analysis of COVID-19 related challenges and the role of technology use among

rural community residents. *BMC Public Health* 22, 1 (27 Apr 2022), 845. doi:10.1186/s12889-022-13254-1

[5] Matus Formanek and Martin Zaborsky. 2017. Web Interface Security Vulnerabilities of Selected European Open-access Academic Repositories. *LIBER Quarterly* 27, 1 (2017), 45–57.

[6] Mahmudul Hasan, Farhana Zaman Rozony, Md Kamruzzaman, and Md Kazi Shahab Uddin. 2024. Common Cybersecurity Vulnerabilities: Software Bugs, Weak Passwords, Misconfigurations, Social Engineering. *Global Mainstream Journal of Innovation, Engineering & Emerging Technology* 03, 04 (august 2024), 42–57. doi:10.62304/jieet.v3i4.193

[7] International Organization for Standardization. 2018. *ISO/IEC 27000:2018 Information technology – Security techniques – Information security management systems – Overview and vocabulary.* Standard. ISO/IEC.

[8] V. Benjamin Livshits and Monica S. Lam. 2005. Finding Security Vulnerabilities in Java Applications with Static Analysis. In *Proceedings of the 14th USENIX Security Symposium*. USENIX Association, 271–286.

[9] OWASP Foundation. 2025. OWASP Top 10:2025. https://owasp.org/Top10/2025/.

[10] J. S. Rauthan and K. S. Vaisla. 2017. Privacy and Security of User's Sensitive Data: A Viable Analysis. In *Proceedings of the Second International Conference on Research in Intelligent and Computing in Engineering (ACSIS, Vol. 10)*. 67–71. doi:10.15439/2017R45

[11] Felipe Ferreira Sampaio. 2021. Uma análise prática das principais vulnerabilidades em aplicações web baseado no top 10 OWASP. Trabalho de Conclusão de Curso, Graduação em Redes de Computadores, Universidade Federal do Ceará.

[12] State of Javascript/Typescript. 2025. Back-end Frameworks Ratios Over Time. Disponível em: https://2025.stateofjs.com/en-US/libraries/back-end-frameworks/#back_end_frameworks_ratios.

[13] M. Syarifudin, Lilik Widyawati, and Ondi Asroni. 2025. Web Security Vulnerability Analysis and Mitigation Based on OWASP TOP 10. *Journal of Artificial Intelligence and Engineering Applications* 4, 3 (june 2025), 1830–1834.

[14] TCM Security. 2025. OWASP Top 10 Predications for 2025. Disponível em: https://tcm-sec.com/owasp-top-10-prediction-2025/.