Vilniaus universiteto leidykla

Contents lists available at Vilnius University Press

Jaunųjų mokslininkų darbai
2025, vol. 55, pp. 64–73

eISSN 1648-8776
DOI: https://doi.org/10.15388/JMD.2025.55.7

Assessing Vulnerability of Students' Programming Projects: Application of Testing Tools and Estimation of Checklist Effect on Code Quality

Studentų programavimo projektų pažeidžiamumo vertinimas: testavimo įrankių taikymas ir kontrolinio sąrašo įtakos kodo kokybei įvertinimas

Eligijus Andriulionis

Silutes Vyduno Gymnasium
E-mail ari@ari.lt

Simona Ramanauskaitė, Prof. Dr.

Vilnius Gediminas Technical University
E-mail simona.ramanauskaite@vilniustech.lt
https://ror.org/02x3e4q36

Tatjana Balvočienė

Silutes Vyduno Gymnasium
E-mail tatjana.balvociene@vydunas.lt

Summary. Web application security is one of the mandatory elements in system development, however, the proper level of security measures among beginner level programmers is still an issue. This paper examines how security checklists impact the secure development practices and code quality in novice developers, within web application development using the *Flask* framework. In a controlled experiment, four university students were asked to develop a sleep tracking system using the *Flask* web framework, then later asked to improve it by either using a short or a comprehensive security checklist. This research studies how such checklists drive the identification and mitigation of common security vulnerabilities, such as XSS, SQL injection, and poor key management. Using automated and manual code reviews, this study assesses the efficiency of such checklists in improving both security and general code quality, and hence their potential value in academic and professional environments.

Keywords: website security, vulnerability, security checklists, code quality, student projects.

Santrauka. Žiniatinklio programų saugumas yra vienas iš privalomų sistemos kūrimo elementų, tačiau pradedančiųjų programuotojų žinių lygis apie saugumo priemones vis dar yra problema. Straipsnyje nagrinėjama, kaip saugos kontroliniai sąrašai veikia saugaus kūrimo praktiką ir kodo kokybę, kai pradedantieji progra-

Received: 2025-03-02. **Accepted:** 2025-03-20

muotojai kuria žiniatinklio programas naudodami *Flask* karkasą. Kontroliuojamo eksperimento metu keturių universiteto studentų buvo paprašyta sukurti miego sekimo sistemą naudojant *Flask* žiniatinklio karkasą, o vėliau paprašyta ją patobulinti, naudojant trumpą arba išsamų saugos kontrolinį sąrašą. Pristatomame tyrime analizuojama, kaip tokie kontroliniai sąrašai padeda nustatyti ir sumažinti įprastas saugumo spragas, tokias kaip *XSS*, *SQL* injekcija ir prastas raktų valdymas. Naudojant automatines ir rankines kodų peržiūras, šiame tyrime vertinamas tokių kontrolinių sąrašų efektyvumas gerinant saugumą ir bendrą kodo kokybę, taigi ir jų potencialią vertę akademinėje ir profesinėje aplinkoje.

Pagrindiniai žodžiai: tinklalapių saugumas, pažeidžiamumas, saugos kontroliniai sąrašai, kodo kokybė, studentų projektai.

Introduction

As the world continues to digitalize, web applications become not separable from our daily life and work. Vulnerabilities such as Cross-Site Scripting (XSS), SQL injection, and poor key management are among the most common web application security issues and can lead to severe data breaches, loss of user trust, and potential harm to both users and organizations. This issue is compounded partly by the lack of clearly structured guidelines, best practices for the novice programmers. Detailed checklists for system security level improvement and systematic approach for web application security testing might be a way to change the situation and move toward more secure web system development.

This research aims to investigate how security checklist usage affects system security developed by a novice programmer. It is focused on two main research questions:

- RQ1: How do publicly available results of web system vulnerability scanning tools match experts' evaluation on the estimation of the *Flask* project web system security level?
- RQ2: How efficient are web system security assurance checklists among novice programmers?

In this research, we will concentrate on python-based web system development using the *Flask* micro web framework. This specific technology was selected taking into account the constantly increasing popularity of Python language and students' orientation to learn this technology and apply it for web system development.

Related Work

Checklist-Based Code Reviews

Studies have explored the impact of checklists on detecting security vulnerabilities in code reviews. Braz et al. (2022) found that simply instructing reviewers to focus on security increased vulnerability detection eightfold, while adding a checklist had little effect, highlighting the importance of mindset.

In education, Chong et al. (2021) analyzed 1,791 checklist questions from 394 students, finding that while students could anticipate defects, misconceptions about code reviews remained, emphasizing the need for better instruction.

Checklists serve various purposes beyond security. Fedele et al. (2024) developed the ALTAI checklist for ethical AI. However, Su (2024) notes that while checklists and testing tools are essential for learning secure development, they must be part of a structured study process.

Automated Vulnerability Assessment Tools

Web vulnerability scanning relies on Static (SAST) and Dynamic (DAST) testing. SAST analyzes code without execution to detect early-stage flaws, while DAST tests runtime behavior to find execution-related vulnerabilities.

Esposito et al. (2024) found SAST tools highly precise but limited in scope, emphasizing the need for complementary methods. Bennett et al. (2024) noted SAST's role in early detection but highlighted challenges in adoption and configuration, especially for novice developers.

DAST remains essential for runtime security but faces automation and coverage challenges (Sutter et al., 2024). Research suggests combining SAST and DAST improves detection and reduces false positives (Nunes et al., 2024), requiring continuous adaptation to evolving threats (Kumaran et al., 2024).

Research Methodology

The research was designed (see Fig. 1) to gather the needed data to answer research questions and guarantee that the data are trusted. The research was started by analyzing different sources for building main and extended checklists for security assurance level of a *Python* and *Flask* micro web framework-based project.

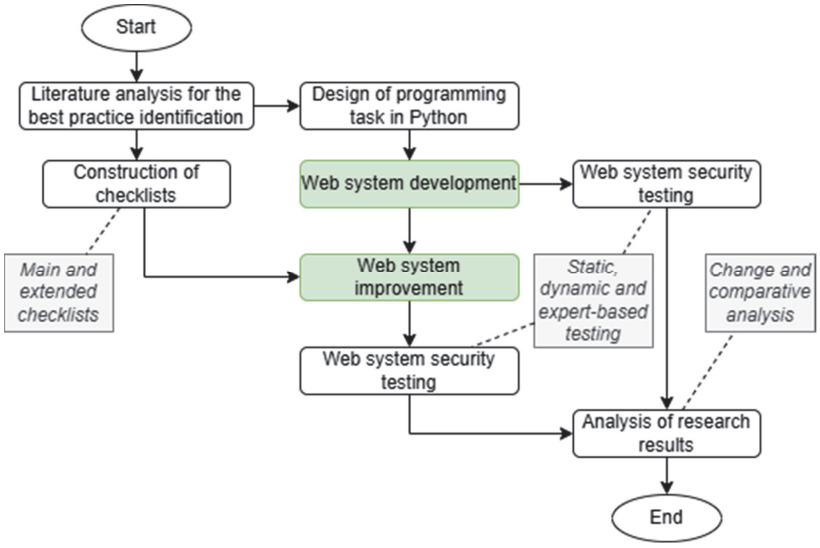

Fig. 1. **Main flow of the research**

The extended checklist serves as a detailed framework, addressing a broad range of vulnerabilities from out-of-bounds access to *XSS* attacks. It is applicable to both front-end and back-end development, as well as general administration practices. Meanwhile, the main version of the checklist was created retaining the essential security considerations while minimizing content. The checklists were composed taking into account that they will be used with *Flask* projects, however, presenting them in a more general way, making sure that the checklist is suitable for any web system project. The key features of each checklist are:

- Main checklist[1]: This 32-item list (2.4 KB in size) provides quick, concise instructions to help prevent common vulnerabilities.
- Extended checklist[2]: This 56-item list (10 KB in size) provides more detailed guidance, with specific questions about the code and suggestions for preventing potential issues. Each item also lists relevant vulnerabilities.

A programming task was developed separately from the checklist to ensure neutrality. It focused on common web vulnerabilities, including user management and CRUD operations, while keeping the project small and relevant to early-year students.

The task was given to 3rd-year IT students with prior desktop programming experience but limited web development exposure. As an optional assignment, it introduced *Flask* and *Python* for web development. While students had no formal cybersecurity training yet, a dedicated course was planned for later, after they gained experience with various programming languages and platforms.

The research with students was conducted in two main stages (marked with green background in Fig. 1):

1. Students developed a *Flask* web app without guidance to assess their independent coding decisions. Only four projects were submitted.
2. Students were then split into two groups (two per group) and given different security checklists to review and refine their code without external feedback. This tested the impact of structured security guidance. All four students submitted revised projects and shared feedback on checklist usability.

In both stages, methodological rigor was ensured to minimize methodological variability in the study's results. Each participant received only one type of checklist, without additional information other than that contained in the checklists. Despite the fact that the number of participants is very limited, the students' skills in this course were of very similar level, with no distinction in one group or another.

The received students' projects were deployed and tested for security vulnerabilities. The same testing procedure and tools were used for the initial and updated students' projects. The testing stage used a mixed-methods approach to analyze students' programming practices, incorporating both manual and automated analyses. Automated tools such as

[1] https://git.ari.lt/ari/research-school-2024/src/branch/main/writeups/mini-checklist.md
[2] https://git.ari.lt/ari/research-school-2024/src/branch/main/writeups/checklist.md

Pyright[3], *Pylint*[4], *Bandit*[5], and *OWASP ZAP*[6] were used to identify potential issues within the code. The first ones are SAST, while the last ones are DAST tools, freely available for personal application. At the same time, the main focus was on manual web application security analysis, which allowed for more context-aware insights from the students' code submissions. Manual analysis is particularly effective at detecting subtle 'code smells' and security vulnerabilities that automated tools may overlook.

Automated vulnerability testing used the latest tools (as of January 2025), while manual analysis involved code review and system testing. Each project received a list of strengths and weaknesses, with expert evaluations summarizing checklist adoption.

The final stage compared project vulnerability levels using five evaluation tools, assessing changes between initial and updated versions to measure checklist impact.

Research results

Summary of Research Results

As shown in the Fig. 2, in the first stage, the students generated 877 lines of *HTML*, 711 lines of Python, and 158 lines of *CSS*. The second stage slightly exceeded these figures, with 888 lines of *HTML*, 971 lines of Python, and 158 lines of *CSS*.

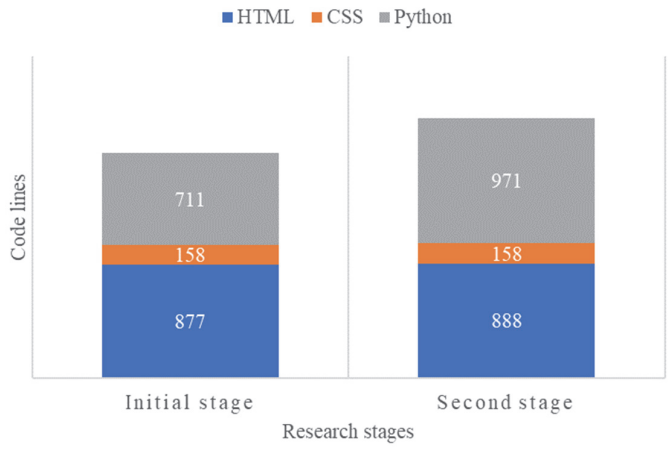

Fig. 2. **Code change in the first and second stages of the research**

The growth ratios across the two stages reveal a modest increase of 0.6% for *HTML* and a more significant rise of 15.5% for Python, while *CSS* remained unchanged with a growth ratio of 0%. Overall, the projects' size increased by 22.2%.

[3] https://pypi.org/project/pyright/
[4] https://pypi.org/project/pylint/
[5] https://bandit.readthedocs.io/en/latest/
[6] https://www.zaproxy.org/

In the initial phase of the experiment, students who implemented the sleep monitoring system without security guidance demonstrated various vulnerabilities characteristic of insecure programming practices. Simple mistakes were prevalent, from hard-programming credentials, lack of input validation, to poor handling of sensitive data. The review showed that these issues represented serious threats, which in a real-world usage scenario would have serious consequences.

The testing results are summarized in Table 1, listing the numeric values of the testing results, which can be used for easier comparison.

Table 1. **Summary of project vulnerability analysis results**

Stage	Checklist	Student	SAST			DAST	Manual
			***Pyright*: 1.1.391**	***Pylint*: 3.3.3**	***Bandit*: 1.8.2**	***OWASP ZAP*: 2.15.0**	**Expert**
Initial project	NA	A	65 errors	8.17/10	2 low, 0 medium, 1 high	3 medium, 4 low, 4 informational	14 issues, 6 strengths
		B	33 errors	7.61/10	1 low, 0 medium, 1 high	2 medium, 2 low, 0 informational	13 issues, 5 strengths
		C	72 errors	6.59/10	1 low, 7 medium, 1 high	3 medium, 3 low, 3 informational	15 issues, 3 strengths
		D	65 errors	5.87/10	2 low, 0 medium, 1 high	4 medium, 3 low, 4 informational	11 issues, 5 strengths
Updated project	Main	A	69 errors	8.00/10	2 low, 0 medium, 0 high	4 medium, 4 low, 5 informational	4 fixes, 7/32 checklist
		B	59 errors	6.47/10	1 low, 0 medium, 0 high	5 medium, 4 low, 7 informational	8 fixes, 26/32 checklist
	Extended	C	96 errors	7.70/10	1 low, 0 medium, 0 high	5 medium, 7 low, 7 informational	9 fixes, 17/56 checklist
		D	107 errors	4.98/10	1 low, 0 medium, 1 high	5 medium, 7 low, 7 informational	10 fixes, 38/56 checklist

Baseline data show that indeed all the projects had very serious security problems before the application of the security checklists. The average number of original projects was 58.75 errors per project according to *Pyright*. Moreover, for *Pylint*, the average rating of the original projects was 7.06/10, and the lowest score belonged to D, which was 5.87/10. The *Bandit* scan also revealed an average vulnerability count of 0 undefined, 2 low, 2 medium, and 1 high in severity for the four projects. The *OWASP ZAP* automated scan reported a total average of 3 medium, 3 low, and 3 informational vulnerabilities per project.

Analysis of Quantitative Security Analysis

One of the most pervasive issues in the projects during the initial stage of the research was the poor handling of sensitive data: hard-coded credentials, static secret keys, and improperly handled passwords. These were present in all four projects and directly betrayed the open security model by exposing sensitive data. Most of these projects adopted more secure practices after the application of the security checklists in the second stage, updating the project based on the checklist. Even though the projects did not show major improvement in the mere storage of the credentials, the credentials were separated out of most of the code. Proper password hashing was also implemented for more secure password storage by only storing the hash digest of the password. The introduction of a secure password hashing function significantly mitigated the risks from plaintext password storage and weak hashing algorithms like *MD5*.

Input validation was another critical area that needed much attention because initially, most of the projects did not validate the inputs given by users. This could result in possible injection attacks like *SQL* injection, Cross-Site Scripting, and other injection or input validation attacks. After following the security checklists, the projects improved their input validation practices significantly, which also improved the error handling as well.

Another common weakness was the lack of *CSRF* protection in the original projects. This made the applications vulnerable to *CSRF* attacks, where malicious users could trick authenticated users into performing unintended actions. After the implementation of the checklist, 50% of the projects in the second stage added Anti-*CSRF* tokens to their web applications using *Flask-WTF*, hence mitigating this threat. Some projects also implemented a content security policy to avoid *XSS* attacks and make sure that only trusted resources could run within the web application in the front-end.

Among the key takeaways for the students from the manual reviews of these projects, there came out the much-improved default settings of Flask which were addressed. Moreover, secret keys and credentials were not only stored in a more secure way, but performance was also optimized by reusing database connections. The improvements and quality of them also directly correlated to how thoroughly the checklists were followed. Although far from perfect, project D was substantially more secure compared to its initial state and showed a very good understanding of both security principles and best coding practices in the second stage.

Analysis of Checklist Application Efficacy

The hypothesis was that the usage of checklists helps to increase the security level of beginner-level programmers' projects. Meanwhile, the comparison of the results of automated testing tools indicates that the ratio of system vulnerability level for most of the tools increased (see Fig. 3). Once the security fixes had been applied, several of the statistics shifted significantly. In many cases, the automated scan results were worse or no better than the original findings. While counter-intuitive, this is not unusual as systems become increasingly complex, true review requires sophisticated understanding of context to understand the code and make educated assessments on the project's source

code. However, *Bandit's* results improved for most of the projects, which correlates well with the mitigated security flaws.

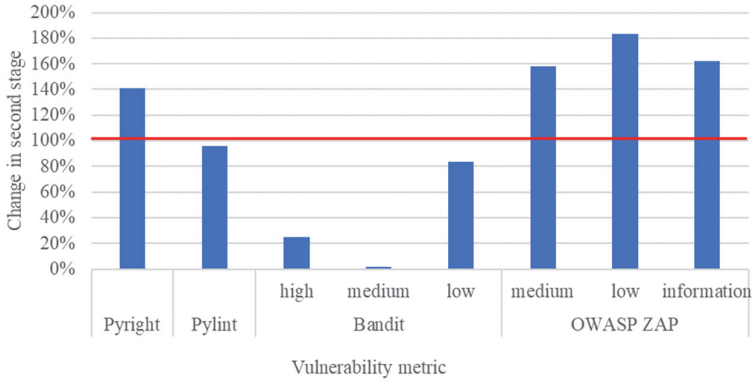

Fig. 3. **Change of vulnerability metrics ratio between the first and second stage**

The security enhancements recommended by the checklists added complexity to the applications, and complexity often begets an increase in lint-time errors. The added complexity is not only supported by the lint-time error count related to code quality, but also the 20% code size increase as demonstrated above. It is worth pointing out, though, that even while these are not directly security-related issues, better code quality −in the sense of clean, organized, and optimized code −usually means fewer security bugs and more efficient mitigations; a fact to which the results of this project proved no exception.

To go deeper into the project vulnerability, the change in these two stages, the most common vulnerability groups were highlighted. For each of them, the percentage of projects having this vulnerability was marked. This was done both in the initial as well as the second stage (see Table 2).

Table 2. **Summary of manual code analysis based on vulnerability categories**

Vulnerability group	**Percentage of project with this vulnerability**	
	Initial stage	**Second stage**
Static secret keys	100%	25%
Lack of input validation	100%	0%
XSS vulnerabilities	100%	0%
CSRF and CSP vulnerabilities	100%	50%
Password storage problems	75%	0%
SQL injection vulnerabilities	25%	0%
Unsafe cryptography	25%	0%
Code quality	100%	25%
Other vulnerabilities and configuration flaws	75%	25%
Average among all categories	78%	14%

The manual analysis results indicate quite a drastic increase in the project security level. During the second stage, still half of the projects contained *CSRF* and *CSP* vulnerabilities, one project had static secret keys, the code quality was not proper, and other configuration flaws existed. Meanwhile, all other vulnerabilities were eliminated. The change in vulnerability level match *Pylint* and *Bandit* indicated changes in vulnerability level.

Analyzing how existing automated security testing tools obtained while testing the results relate to the manually identified number of security issues, the linear regression model indicates that the highest importance is assigned to the number of high (0.34) and medium (0.38) level errors in the Bandit tool. Meanwhile, the only remaining parameter with positive feature weight is *Pyright* score (0.01). All the remaining metrics have a negative weight, where *ZAP OWASP* information notices reach the highest negative weight (-0.45). This indicates that the expert's evaluation is still a core element in security evaluation. However, application of automated tools motivates to look for all possible security issues or code flaws.

Analyzing what exactly was changed in each project, it is visible that the main checklist is effective – 12 fixes were added by the students A and B, while the students C and D added 19 fixes to their code, which indicates that the extended checklist gave a wider coverage of the vulnerabilities to take into account. The most significant improvements directly addressed major vulnerabilities on sensitive data management, input validation, and password security. The projects that followed fewer items in the checklist demonstrated very modest improvements. Meanwhile, basic security measures were implemented, like password hashing and input validation, the lack of comprehensive changes resulted in only minimal improvements. In contrast, project D applied 38 out of 56 checklist items and showed the most robust improvement: more secure session management, improved error handling, and enhanced cryptographic as well as general security and programming practices. This leaves us with a conclusion that our checklists had positive outcomes in security, however, it does not mean that it improved the code quality overall.

Discussion and conclusions

After using the security checklists, there was a marked improvement in both security stance and code quality. Students employing the complete checklist achieved more improvements in their code than students who employed the abbreviated checklist. Individuals who utilized the more exhaustive checklist were also in a position to identify weaknesses and resolve them accordingly. This finding suggests that the role of long, formalized advice plays a significant part in instilling a sense of security best practice in novice developers. Consequently, most security weaknesses that appeared at the initial phase of development were largely mitigated.

However, the study also uncovered a latent trade-off between security enhancements and more complex code. Meanwhile, the participants working with the checklists showed significant security gains, this came at the cost of more complicated code. Inclusion of security aspects led to increased lint-time bugs and an approximated 20% code growth.

This new complexity can cause issues for maintenance and long-term quality, particularly for new programmers who do not yet fully understand standard security practices.

The results highlight the importance of balancing security enhancements with code simplicity. The results suggest that security checklists can significantly improve the security practices of novice developers, but proper care must be taken into account as to how these are affecting the overall quality and maintainability of the code. Encouraging developers to adopt a mindset that aligns security practice with clean programming standards can potentially lead to more enduring software development outcomes.

In conclusion, the findings of this study indicate the effectiveness of security checklists in guiding novice programmers to attain better secure programming practices. The added code complexity that accompanies it, however, necessitates ongoing research and debate about the most effective ways of deploying security without impairing code maintainability. Future work should go towards improving checklist designs and injecting best practices which consider security as well as code simplicity to ensure that starting developers can yield secure, good quality software with minimal complexity to overwhelm them.

References

Bennett, G., Hall, T., Counsell, S., Winter, E., & Shippey, T. (2024, October). Do Developers Use Static Application Security Testing (SAST) Tools Straight Out of the Box? A large-scale Empirical Study. In Proceedings of the 18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (pp. 454-460).

Braz, L., Aeberhard, C., Çalikli, G., & Bacchelli, A. (2022, May). Less is more: supporting developers in vulnerability detection during code review. In Proceedings of the 44th International conference on software engineering (pp. 1317-1329).

Chong, C. Y., Thongtanunam, P., & Tantithamthavorn, C. (2021, May). Assessing the students' understanding and their mistakes in code review checklists: an experience report of 1,791 code review checklist questions from 394 students. In 2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET) (pp. 20-29). IEEE.

Esposito, M., Falaschi, V., & Falessi, D. (2024, June). An extensive comparison of static application security testing tools. In Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering (pp. 69-78).

Fedele, A., Punzi, C., & Tramacere, S. (2024). The ALTAI checklist as a tool to assess ethical and legal implications for a trustworthy AI development in education. Computer Law & Security Review, 53, 105986.

Kumaran, U., Sree, P. S., Udaya Sree, S., Sowgandhi, V. K., & Balasubramanian, S. (2024, April). Web Vulnerability Scanner. In International Conference on Advances in Information Communication Technology & Computing (pp. 193-207). Singapore: Springer Nature Singapore.

Nunes, P., Fonseca, J., & Vieira, M. (2024). Blending Static and Dynamic Analysis for Web Application Vulnerability Detection: Methodology and Case Study. IEEE Access.

Su, J. M. (2024). WebHOLE: Developing a web-based hands-on learning environment to assist beginners in learning web application security. Education and Information Technologies, 29(6), 6579-6610.

Sutter, T., Kehrer, T., Rennhard, M., Tellenbach, B., & Klein, J. (2024). Dynamic security analysis on android: A systematic literature review. IEEE Access.