SCIENCE SET
OPEN ACCESS PUBLISHERS

Review Article

World Journal of Artificial Intelligence and Robotics Research

Analysis of Open Source Security Threats

Vernes Vinčević

University of Vitez VITEZ, Faculty of Information Technologies, Školska 23, 72270 Travnik, Bosnia and Herzegovina

***Corresponding author:** Vernes Vinčević, University of Vitez VITEZ, Faculty of Information Technologies, Školska 23, 72270 Travnik, Bosnia and Herzegovina.

Submitted: 26 December 2025 **Accepted:** 02 January 2026 **Published:** 05 January 2026

Citation: *Vičnević, V. (2026). Analysis of open-source security threats. Wor Jour of Arti inte and Rob Res, 3(1), 01-06.*

Abstract

This paper analyzes security threats specific to open-source software. Partic- ular attention is given to vulnerabilities caused by the open nature of the code, lack of centralized control, and dependence on external libraries. Case studies of known vulner- abilities and protection methodologies used in open-source communities are presented. It will discuss the methods that open source communities apply to prevent security breaches, and how automated tools contribute to the early detection of problems. The aim of the paper is to provide a clear overview of the threats and present best practices that can increase the security of open source software projects. The research methods applied include literature analysis, case studies, and comparative analysis of security practices.

Keywords: Analysis, Threats, Open Source Code, Protection, Security, Software, Security Incidents, Vulnerabilities.

Introduction

In the last few years, open source software has become the foundation of many technological solutions around the world. Due to its availability, flexibility and joint development, this kind of software has enabled rapid progress in numerous fields, including web development, cloud computing and artificial intelligence. However, the open nature of the code has also opened up new security challenges, because anyone has the possibility of accessing the source code - including potentially malicious users.

The security of open source software has become a particularly important topic due to the increasing number of threats and incidents that originate precisely from inadequate control of changes in the code and the lack of standardized security checks. The motivation for processing this topic stems from the need for a better understanding of the specific risks that open source carries, as well as the need to analyze the approaches used to reduce such risks. The issue of security has become crucial not only for developers and system administrators, but also for organizations that rely on open-source solutions.

The paper will analyze the main security threats that occur in open source software, including malicious contributions, vulnerabilities in external libraries, and the lack of systematic control. The paper will analyze security threats in open source software, including their sources, characteristics, and typical consequences. Then, specific cases of security incidents (e.g. Log4Shell and Heartbleed) will be presented, as well as the protection methods that have been implemented in communities after these events. At the end of the paper there is a conclusion summarizing the most important findings and recommendations.

Overview of Important Security Threats in Open Source Software

Open source software provides users with access to the source code, with the freedom to modify and distribute it. While this approach encourages innovation and transparency, it also opens up opportunities for security vulnerabilities, especially in cases where projects do not have a solid quality control and security structure. Security threats in the open-source environment are becoming increasingly significant, given that such software increasingly forms the foundation of modern digital services [1]. According to definitions in the literature [2], information system security includes the protection of data, software components, and user access. Below we present some of the more important security threats.

Malicious Commits

In large open source communities, where contributions come

from different sources, it is possible for an attacker to make malicious changes that appear to be harmless functionality. One of the most famous cases of compromise occurred in 2018 within the Node.js event-stream library, when an attacker managed to inject malicious code with the aim of stealing crypto wallet user data [3]. Such attacks are often called "supply chain attacks", because they target the dependency chain through which malicious content is transmitted to legitimate applications.

Vulnerabilities in Third-Party Dependencies

According to research conducted by Snyk (2021), more than 80% of security vulnerabilities in open-source software come from third-party dependencies, rather than from the direct code of the project itself [4]. These vulnerabilities often go undetected because developers do not check transitive dependencies – libraries that other libraries use.

It is for this reason that an increasing number of projects use automated tools such as Dependabot, Snyk and OSS-Fuzz, which analyze the dependency chain and warn of known vulnerabilities.

Abandoned and Poorly Maintained Projects

According to the analysis, the 2022 attack known as the "Colors/Faker incident" shows how risky unmaintained projects can be. When the author of the popular faker.js library deliberately released a non-functional version as a form of protest, thousands of applications around the world were left non-functional [5].

Without constant control and an active community, open-source projects become vulnerable to unwanted changes, as well as gradual "code rot".

Inadequate Authentication and Authorization

Many software projects, especially smaller and individual ones, do not implement advanced access control mechanisms. Some solutions even contain hard-coded passwords or inadequate encryption of user data [6]. Advanced frameworks such as OAuth2, JWT, and OpenID Connect enable robust identification systems, but their integration requires knowledge and resources that many volunteer projects do not have.

Lack of Formal Security Policies

One of the often overlooked aspects of security in open-source projects is the lack of clearly defined security policies. Security policies are a set of rules and procedures that define how vulnerabilities are handled, who is responsible for addressing them, and what tools are used to monitor security.

Without formal policies, accountability remains unclear, which can lead to delays in responding to threats, lack of coordination within the team, and increased risk of exploitation. The presence of files such as SECURITY.md in repositories helps define these procedures and provides users and contributors with a clear view of the project's security model.

By introducing basic rules for responsible vulnerability disclosure, as well as procedures for reporting and responding to incidents, a project's resilience to attacks is significantly increased.

Threats Through Ci/Cd Flows

Modern open-source projects often use CI/CD (Continuous Integration/Continuous Deployment) workflows to automate testing, building, and distributing software. These workflows often have access to sensitive resources-such as API keys, secret tokens, and permissions to publish packages.

Attackers can compromise '.yml' configurations, add malicious scripts, or misuse access tokens that are not properly secured. As GitHub's research on CI workflow attacks shows [7], attacks on automated workflows are becoming an increasingly common exploitation vector.

Recommended security measures include:

- Limiting CI tool privileges (principle of least privilege),
- Using secrets from security vault systems (e.g. HashiCorp Vault),
- Reviewing all automated scripts and external actions (e.g. GitHub Actions),
- Audit logs of CI job executions.

Social Engineering in Open-Source Communities

Social engineering attacks involve manipulating community members to gain trust and the right to modify code. Open-source communities are particularly vulnerable because they are based on mutual trust and voluntary contributions.

An example is a case where attackers worked imperceptibly to build reputation by contributing for months, only to then insert a vulnerability into a critical project functionality [8]. From a social engineering and communication standpoint (relevant to your earlier work on društveni inženjering), open-source attacks demonstrate how social capital, norms, and human psychology can be weaponized in digital communities, often more effectively than technical exploits.

Typosquatting Attacks

Typosquatting is a type of attack in which an attacker creates a malicious software package with a name that is very similar to the name of a legitimate package, relying on user typing errors. This technique is particularly effective in open-source ecosystems such as npm, PyPI, and RubyGems, where packages are readily available and often installed automatically.

For example, an attacker could create a package called reqeusts instead of requests (a popular Python library), and a user who mistypes the name installs the compromised software. Such packages can contain malware that steals data, sends information to the attacker, or compromises the development environment.

According to a 2021 report by ReversingLabs, more than 3,600 suspicious packages were discovered in the npm repository as part of a survey [9].

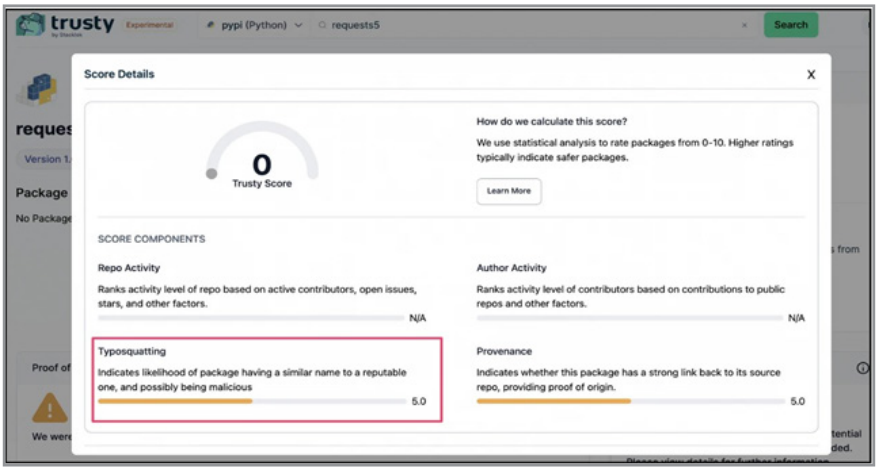

Figure 1: Example of a typosquatting package detection tool – Trusty Score shows the suspicious Python package requests5 with a very low trust level [10].

Figure 1 shows an example of evaluating the requests5 package using the Trusty Score system. The tool uses statistical analyses such as Levenshtein distance and repository activity comparison to estimate the likelihood that the package is the result of a typosquatting attack. Such tools are crucial for preventing development environments from being compromised [11].

Analysis of Practical and Security Methods

Work Methodology

This part of the paper analyzes and presents specific security incidents in open source software and the measures that communities use to mitigate similar risks. Through examples of the Log4Shell and Heartbleed vulnerabilities, it points out the real consequences of security breaches, as well as tools and practices developed to improve protection. Specific examples of known vulnerabilities and protection methodologies used in open source communities are presented. In the preparation of this paper, the methods of analysis of professional literature reviews, compilations, case studies and comparative analysis of existing security approaches were used. Through all these methods, we will demonstrate the importance of research and provide new insights to future researchers through this paper.

Log4shell Vulnerability

At the end of 2021, a critical vulnerability was discovered in the Java library Log4j, known as Log4Shell (CVE-2021-44228). This library is used for logging messages within applications, and the vulnerability allowed an attacker to execute arbitrary code on a remote server without authentication [10]. The incident shook the global IT community, as many applications used Log4j as a dependency, often unknowingly. One of the most dangerous vulnerabilities in the history of software security. An example of an attack could be sending a malicious payload through a web form, or API request, inserting exploits into chat messages, usernames or IoT messages, which results in the theft of data and credentials.

Heartbleed Bug

Heartbleed is a vulnerability discovered in 2014 in the OpenSSL library, which implements the TLS and SSL cryptographic protocols. Due to a flaw in the "heartbeat" mechanism, it was possible for an attacker to read up to 64KB of server memory, which could reveal sensitive data such as passwords and private keys [12]. The vulnerability demonstrated the importance of regular auditing and security testing of core software components. A visualization of how the Heartbleed attack works is shown in Figure 2, where it is seen that the server returns more data from memory than was legitimately requested.

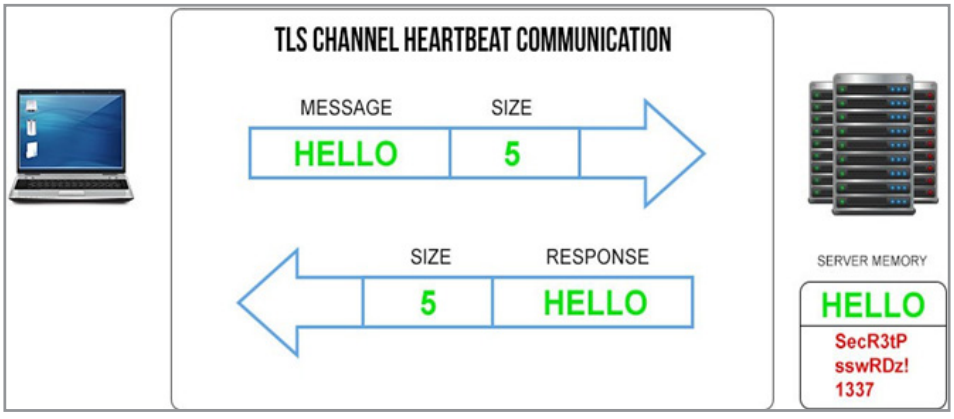

Figure 2: Visualization of the Heartbleed vulnerability – the attacker receives more memory than he requested [12].

The following figure illustrates, analyzes and provides a detailed technical overview of the Heartbleed vulnerability exploitation in Figure 3, where it is seen how the attacker abuses the payload length to gain access to additional memory.

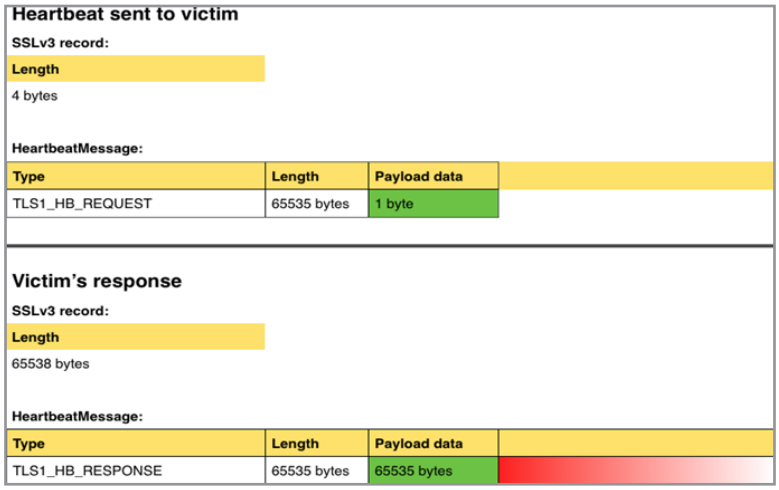

Figure 3: Technical illustration of exploiting Heartbleed vulnerability - difference between actual and reported data length [13].

We conclude from the image that the attacker is manipulating the payload length field in the TLS Heartbeat request, specifying a larger value than the data actually sent. Due to the lack of proper bounds checking in the OpenSSL implementation, the server returns not only the expected payload, but also additional memory chunks from its address space in the response. In this way, the attacker can passively and repeatedly read sensitive data from the server's memory, including private keys, session cookies, and user data, without leaving any visible traces of the attack.

Security Errors in the Configuration

In addition to known vulnerabilities in the code, a large number of security incidents in the open-source environment are caused by incorrect or carelessly adjusted configurations. Such errors are not technical "bugs" in the software, but the result of human carelessness, lack of knowledge of the tools or poor documentation, methodology, etc.

For example, users often leave:

- default passwords (admin/admin),
- enabled test interfaces (e.g. API sandbox open to the public),
- unlimited access to admin panels without authentication,
- excessive privileges to users or services.

Such errors allow an attacker to effortlessly compromise the system, even when the software itself is technically correctly implemented. This is why the importance of security "hardening" and the use of tools for automatic configuration verification, such as OpenSCAP, Lynis and Kube-bench, is emphasized.

Software Tools and Protection

Early detection of security threats in open source projects is based on a combination of specialized software tools and preventive protection methods. Static and dynamic code analysis, tools for automatic detection of vulnerabilities and errors, as well as dependency scanning to identify known vulnerabilities in used libraries play a key role. In addition, the implementation of continuous integration (CI/CD) with built-in security checks, community code reviews, and the use of digital signatures and verification of contributions enable the timely recognition of potentially malicious or insecure code. This multi-layered approach significantly reduces the risk of exploiting vulnerabilities and strengthens confidence in the security of open source software.

Open source communities have developed a number of mechanisms for early detection and prevention of security threats:

- Dependabot – a tool that automatically scans and updates dependencies in GitHub repositories.
- Snyk and SonarQube – analyze security vulnerabilities in code and dependencies.
- SECURITY.md files – define rules for reporting vulnerabilities and security responsible persons.
- Formal code reviews – mandatory in large projects such as the Linux kernel or Kubernetes.

Vulnerability Management Models

Vulnerability management models are a systematic framework for identifying, analyzing, prioritizing, and remediating security vulnerabilities in information systems. These models include continuous vulnerability discovery through scanning and auditing, risk assessment based on the likelihood of exploitation and potential impact, and prioritization using standardized metrics such as CVSS.

In addition to tools that enable automatic detection and remediation of vulnerabilities, organizations are increasingly implementing formal Vulnerability Management Lifecycle models. These models involve a continuous cycle of assessment, identification, classification, prioritization, and resolution of security weaknesses.

The most common stages of vulnerability management are:

- Discovery – scanning the system for known vulnerabilities (e.g., using Nessus or OpenVAS),
- Assessment – analyzing the severity of the vulnerability based on frameworks such as CVSS (Common Vulnerability Scoring System) [14],
- Prioritization – classifying according to risk, exploitability, and business impact,
- Remediation – updating software, changing configurations, removing or replacing vulnerable components,
- Verification – re-scanning and confirming that vulnerabilities have been successfully remediated.

This systems approach helps organizations be proactive in combating security threats, rather than reacting only after an incident. In open-source communities, this model is often implemented through CI/CD flows, where tools like Trivy and Grypa

automatically scan container images and libraries. Good practices have shown that by integrating vulnerability management into organizational processes and security strategy, these models enable the reduction of overall security risk and the strengthening of the resilience of information systems.

The Role of the Community and Institutional Support

Institutional support-provided by organizations, foundations, and regulatory bodies-complements community efforts by offering structured governance, financial resources, legal frameworks, and formal security processes. This includes funding for maintenance, security audits, long-term support, and the establishment of standards and best practices. Together, community collaboration and institutional backing create a balanced ecosystem that enhances trust, improves response to security incidents, and ensures the long-term reliability and security of digital infrastructures.

Open-source projects depend heavily on a community of users and contributors. Security incidents such as Log4Shell and Heartbleed have shown that an informal structure can be an advantage due to rapid response, but also a weakness due to inconsistent security practices. To ensure long-term security, larger organizations and institutions have also become involved. For example, the OpenSSF (Open Source Security Foundation) was formed with the support of Google, Microsoft, and the Linux Foundation with the aim of improving the security of open-source software through [15]:

- funding the maintenance of critical projects,
- developing tools for vulnerability analysis,
- educating contributors about security standards,
- developing guidelines for responsible vulnerability disclosure.

The importance of institutional support is also reflected in increased security budgeting. Governments of some countries, including the USA and Germany, have launched initiatives to evaluate and support the security of open-source packages used in infrastructure.

Discussion and Recommendations for Further Research

The results of the research and analysis confirm that the security of open-source software does not depend solely on the quality of the source code, but on the broader socio-technical ecosystem in which the software is developed, maintained and used. The analyzed cases of Log4Shell and Heartbleed clearly indicate that even projects with a large number of users, a long development history and a strong reputation can contain critical vulnerabilities that remain undiscovered for years.

Of particular note is the fact that both vulnerabilities are the result of insufficiently rigorous validation of input data and assumptions about benign user and environment behavior. This points to a structural problem in open-source development, where functionality and performance are often prioritized over security aspects, especially in the early stages of development.

The discussion also shows that technical measures, such as tools for static and dynamic code analysis, although necessary, are not sufficient on their own. The lack of formalized procedures for vulnerability management, weak integration of security into CI/CD flows and limited resources for long-term maintenance represent significant risk factors. In this context, institutional initiatives such as OpenSSF play a key role in standardizing security practices and providing support to the community.

Based on the analysis, it can be concluded that open source security requires a holistic approach, which connects technical, organizational and human factors, and that trust in open-source solutions is built not only by the transparency of the code, but also by the maturity of the processes behind it.

Through research, the analysis has shown that security in open-source software depends not only on the technology, but also on the social structure of the community, the tools used and the level of threat awareness. As such software is increasingly used for commercial and government purposes, the need for better security practices, automation of protection and education of contributors is growing. According to Jovanović [16], the most effective protection measures require a combination of technical solutions and user education.

Directions for Further Research

Based on the conducted research, it is possible to identify several directions for future scientific and professional research:

- Empirical analysis of the effectiveness of security tools further research can be focused on quantitatively assessing the success of tools for static analysis, dependency scanning and automated monitoring in real open-source projects.
- The role of governance and funding in the security of open-source projects of particular research interest is the relationship between institutional support, stable funding and reducing the number of critical vulnerabilities in key open-source libraries.
- Human factors and social engineering in the open-source ecosystem

Additional research is needed on how trust, reputation and communication within the community can be abused, but also how they can be used to strengthen security.

In general, further research should aim to deepen the understanding of open-source security as a shared responsibility, where technical solutions must be supported by organizational structures, institutional mechanisms, and continuous education of all participants.

Conclusion

The paper analyzes the most significant security threats in open source software, as well as specific incidents that have had a global impact on information security. It is shown that open source, while bringing numerous advantages in terms of transparency, collaboration and innovation, also entails complex security challenges that require a systematic and long-term approach. Threats such as malicious contributions, vulnerabilities in dependencies, unmaintained projects, weak authentication mechanisms and attacks on CI/CD flows confirm that security is not guaranteed by the openness of the code itself, but by the quality of the processes that accompany its development and maintenance.

Through the analysis of cases such as Log4Shell and Heartbleed, it is clearly shown how serious the consequences can be when basic security principles are ignored, including access control,

input validation, dependency auditing and proper system configuration. These incidents point to the high degree of interdependence of modern software systems and the fact that failures in one widely used component can have a chain effect on a large number of organizations and users around the world.

The paper also points to the growing importance of security automation tools, such as Dependabot and Snyk, as well as the application of formal vulnerability management models that enable earlier detection and more efficient remediation of security weaknesses. The role of institutional support through initiatives such as OpenSSF is particularly emphasized, which contribute to strengthening security culture, standardizing practices and providing resources for maintaining critical open-source projects.

Despite progress in technical and organizational security measures, the paper also identifies open issues, especially in the areas of contributor education, social engineering prevention and long-term sustainability of projects without stable institutional or financial support. These challenges indicate the need for further development of security frameworks that are adapted to small and medium-sized open-source projects, as well as the development of advanced tools for early detection of threats in distributed software development environments. Ultimately, it can be concluded that the security of open source software is a collective responsibility of all actors involved in its life cycle – developers, communities, organizations, users and institutions. Only through a combination of technical solutions, formalized processes, continuous education, and institutional support is it possible to build long-term trust in open-source solutions and ensure their secure implementation in modern information systems and software solutions.

References

1. Riley, P. (2020). The opensource risk: Managing vulnerabilities in open software projects. Journal of Software Security, 8, 22–34.
2. Mašetić, S. (2019). Information systems security. Faculty of Electrical Engineering, University of Sarajevo.
3. GitHub Security Lab. (2018). Analysis of the event-stream incident. Retrieved from https://github.com/dominictarr/event-stream/issues/116
4. Snyk Ltd. (2021). The state of opensource security. Retrieved from https://snyk.io/opensourcesecurity-2021
5. ZDNet. (2022). Opensource sabotage: What happened to faker.js and colors.js? Retrieved from https://www.zdnet.com/article/open-source-sabotage-what-happened-to-faker-js-and-colors-js/
6. OWASP Foundation. (2023). Authentication cheat sheet. Retrieved from https://cheatsheetseries.owasp.org/cheatsheets/Authentication_Cheat_Sheet.html
7. Pellegrino, G. (2022). CI/CD attacks in the wild. Retrieved December 9, 2025, from https://securitylab.github.com/research/github-actions-preauth-rce/
8. Perlroth, N. (2021). This is how they tell me the world ends: The cyberweapons arms race. Bloomsbury Publishing.
9. ReversingLabs. (2021). Typosquatting malware infects npm ecosystem. Retrieved October 12, 2025, from https://www.reversinglabs.com/blog/typosquatting-malware-infects-npm-ecosystem
10. Stacklok. (2023). Detecting typosquatting attacks on open-source packages. Retrieved November 30, 2025, from https://stacklok.com/blog/detecting-typosquatting-attacks-on-open-source-packages-using-levenshtein-distance
11. Frontera Marketing. (2014). The ultimate Heartbleed guide for non-techies. Retrieved from https://fronterahouse.com/blog/ultimate-heartbleed-guide-for-non-techies/
12. Williams, C. (2014). Anatomy of OpenSSL's Heartbleed: Just four bytes trigger horror bug. Retrieved December 1, 2025, from https://www.theregister.com/2014/04/09/heartbleed_explained/
13. National Institute of Standards and Technology. (2023). CVSS scoring explained. Retrieved December 3, 2025, from https://nvd.nist.gov/vuln-metrics/cvss
14. Pavlić, M. (2011). Information systems. Zagreb: School Book.
15. Sekaran, U. (2016). Research methods for business. New York, NY: John.
16. Jovanović, R. M. (2020). Information security. Faculty of Electrical Engineering, University of Belgrade.