International Journal of Science and Research (IJSR)
ISSN: 2319-7064
SJIF (2022): 7.942

The Importance of Fixing Transitive Vulnerabilities in Java Libraries

Frolikov Evgenii

Team Lead - Cloud Linux, Turkey Mersin
Email: *frolikov123[at]gmail.com*

Abstract: ***In the context of rapidly evolving technologies, ensuring software security is becoming an increasingly relevant task. One critical aspect of this task is addressing transitive vulnerabilities in Java libraries. Transitive vulnerabilities arise when libraries depend on other libraries that contain vulnerabilities, creating a complex web of interdependencies. These vulnerabilities can be difficult to detect and pose significant risks to application security. This paper examines the causes of transitive vulnerabilities, their impact on software security, and methods for their detection and mitigation. The importance of using automated dependency analysis tools, regularly updating libraries, and maintaining strict version control is emphasized. Additionally, measures to minimize risks associated with transitive vulnerabilities are discussed, including continuous security monitoring and the implementation of best practices in software development.***

Keywords: vulnerabilities, java libraries, transitive vulnerabilities, correction of transitive vulnerabilities, programming

1. Introduction

In a rapidly evolving world, achieving excellence demands continuous progress. Recent reports on the state of the software supply chain indicate that developer productivity significantly increases when they have access to superior tools and high-quality open-source components. This, in turn, enhances the security and quality of the products [1].

In this context, the Java programming language and its numerous libraries play a central role, being used to develop a wide range of software solutions, from mobile applications to enterprise systems. However, as the functionality of Java libraries expands, vulnerabilities inevitably arise, which can be exploited by attackers for various attacks [2].

Developing secure Java applications, free from vulnerabilities, is the best way to ensure their reliability and protection against threats. Integrating security measures into the development process helps prevent the creation of vulnerabilities; addressing potential vulnerabilities during the development stage is significantly less time-consuming and resource-intensive than fixing them once they are deployed in a production environment.

Moreover, the interest in this research topic is driven by the patented technology for monitoring vulnerabilities in Java libraries, patent number 508178618.

This paper aims to highlight the importance of addressing transitive vulnerabilities in Java libraries.

2. Literature Review

There is a vast amount of research focused on analyzing the source code of applications written in various programming languages. These studies often aim to compare the applicability of different programming languages for solving specific tasks in physics and mathematics, emphasizing ease of use, expressive power, and minimizing errors during software development.

A significant portion of the research concentrates on collecting data from the GitHub platform about the usage of different programming languages. These studies analyze the number of lines of code, the number of projects a developer participates in, the speed of bug fixes upon request, and other parameters. According to Geiger R.S., the primary source of identifying and reporting vulnerabilities and bugs in code is the user community. This is not surprising, given that GitHub is the largest platform for hosting both proprietary and open-source code.

The scientific community is equally interested in the quality of code and in comparing programming languages in terms of their susceptibility to vulnerabilities in the source code. For instance, Ray B. A explores the code quality on the GitHub platform for various programming languages. His methodology involves identifying keywords associated with fixable vulnerabilities in commit logs. This approach does not provide an objective picture of the prevalence of vulnerabilities, as it only allows for the collection of data on vulnerabilities after the fact, which does not help in building a predictive model for assessing code quality. Similar methodologies are used in the works of Gyimesi P. and Kapur R.

A review of both domestic and international sources also highlights the lack of a code quality assessment system that can be used by non-specialists. This paper will examine the importance of transitive vulnerabilities in Java libraries [3].

3. Materials and Methods

All software contains vulnerabilities that can arise at various stages of its lifecycle. Eliminating all vulnerabilities in the code is a complex task; however, their number can be significantly reduced. The situation becomes more complicated when dealing with third-party software, as fixing vulnerabilities in borrowed libraries or frameworks is a labor-intensive process [4].

Initially, statistical test results were analyzed to identify correlations between quality metrics and code vulnerability.

Such a relationship could indicate a strong correlation between a specific metric and vulnerability. For example, if a high SourceRank is associated with a low level of vulnerability, developers could consider SourceRank when selecting new components.

Unfortunately, no such direct dependency was found. None of the quality metrics showed even a moderate correlation with vulnerability levels. To explore this further, a series of experiments were conducted.

The first experiment involved providing the model with data on OpenSSF Criticality, Security Scorecard, Libraries.io SourceRank, MTTU, and Popularity metrics. The results showed that combining various quality metrics can be quite effective in identifying vulnerable projects. When all quality metrics were used, the model demonstrated an impressive precision and recall of 95.5%. This means that in 95.5% of cases, the model correctly identifies projects with known vulnerabilities, indicating that the quality metrics of a project indeed reflect important factors affecting security.

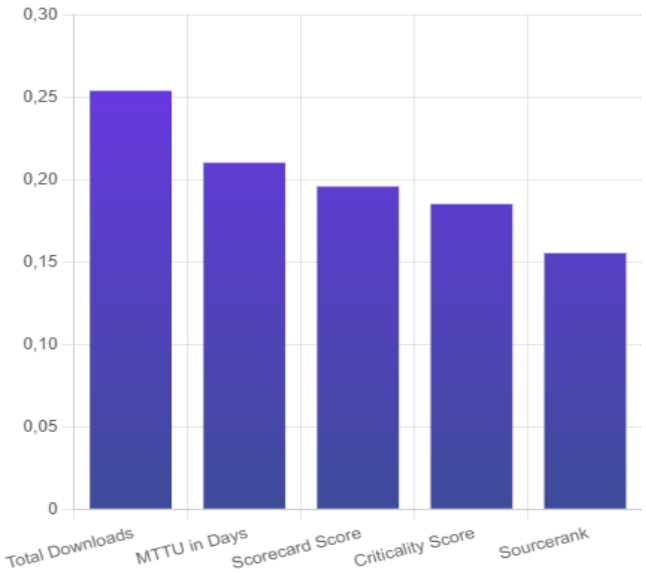

Figure 1: The relative importance of quality indicators in the research model [5].

Figure 1 shows the relative importance of each characteristic in the model. To calculate the feature importance, a model was built excluding it, and the performance decline was measured. The total number of downloads was the most important attribute, which is not surprising given the correlation between popularity and vulnerability. MTTU was the second most important, indicating that dependency update behavior signals the quality of the project. Next in importance were the Scorecard, Criticality, and SourceRank metrics.

Since OpenSSF publishes individual checks that are considered in their metric system, we were able to test how well a model based solely on these advanced software development practices could correctly identify projects with known vulnerabilities. When provided with individual features for the machine learning process, we achieved an accuracy of 89% (precision 86% and recall 87%) for this metric-based model, which is not much lower than the performance of the model using aggregated metrics. This demonstrates that the individual metrics in the scorecard system are very useful on their own.

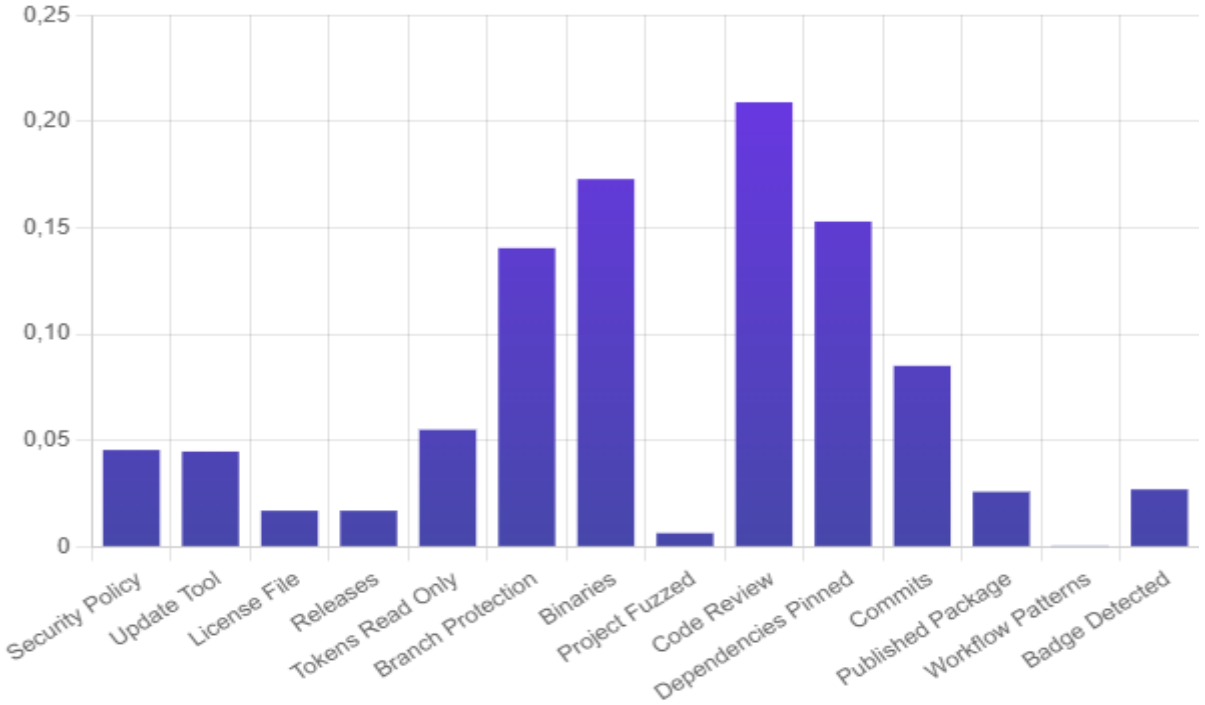

Figure 2: The elements most useful for identifying vulnerable projects [5].

An analysis of significance was also conducted to determine which practices are most critical. Figure 2 presents information on which elements of the security score system proved most useful for identifying vulnerable projects. Unsurprisingly, code review was the top factor in significance. Code review has long been recognized as one of the most effective practices for improving code quality. The use of binary files represents an alternative attack vector, reduces transparency, and complicates code auditing; therefore, their absence in the repository was the second most important factor. Pinning dependencies was the third most significant factor, highlighting the importance of dependency management in ensuring software security. Branch protection was the next significant factor, as it ensures a formal process for approving code changes, which complements the practice of code review [5].

4. Causes and risks of transitive vulnerabilities

Transitive vulnerabilities arise from the use of libraries that, in turn, depend on other libraries. This creates a complex web of dependencies, where a vulnerability in one library can affect numerous other libraries and applications. The main causes of transitive vulnerabilities include:

1) Lack of transparency in dependencies.
2) Absence of version control.
3) Inadequate testing of libraries for security vulnerabilities.

Thus, if a vulnerability is discovered in a library, all applications using it are potentially at risk. In this chain, transitive dependencies—those libraries on which our main libraries depend—play a particularly important role due to their hidden and pervasive nature. The main risks associated with transitive vulnerabilities are:

1) Lack of control. Direct dependencies are usually chosen with great care: after checking the documentation, popularity, update frequency, and even code analysis. In contrast, transitive dependencies may not undergo such thorough analysis since they come bundled with the main packages. This difference in scrutiny means that issues in transitive dependencies can go unnoticed for a long time.
2) Entry points for malicious code. Since transitive dependencies are often overlooked, they become an attractive target for attackers. They do not need to attack a popular package directly; they can choose a less-known package that the popular one depends on. If such a transitive dependency is compromised, anyone using the main package may unwittingly introduce malicious code into their projects.
3) Version management complexity. Transitive dependencies may require specific versions to function correctly. However, several direct dependencies might rely on different versions of the same transitive dependency, leading to version conflicts and unforeseen issues.
4) Licensing problems. Not all dependencies may have the same licensing terms. The main dependency might be licensed under terms compatible with the project, but its transitive dependencies might not meet these terms, leading to legal issues [6].

5. Methods for detecting and mitigating transitive vulnerabilities

To effectively manage transitive vulnerabilities in Java libraries, the following methods are recommended:

a) Utilizing automated dependency analysis tools. Employ tools such as Maven Dependency Plugin and OWASP Dependency-Check to automatically analyze dependencies and detect vulnerabilities [7]. To fix a transitive library in Maven, override the transitive dependency by adding the dependency with the appropriate version as a direct library [8].

Analyzing statistical data on libraries in MavenCentral is of significant interest. However, there are numerous other repositories, such as JCenter, that contain unique libraries not found in MavenCentral. The number of such libraries and the projects that use them can be quite substantial (Table 1).

Table 1: Analysis of Statistical Data [9].

Repository	Total Domains	Vulnerable Domains	Percentage Vulnerable
Maven Central	26,163	3,71	14.18%
GitHub	7,523	291	3.86%
Overall	33,938	6,17	18.18%

The greatest threat comes from transitive dependencies, as it is difficult to track which additional libraries are included in a project through direct dependencies. Next, we will consider the public and repository response to the issue. On January 19, 2024, Sona type released a statement outlining the measures taken to mitigate the impact of this attack:

- According to their statement, DNS verification is only performed upon the first publication. Subsequently, to regain account control, contacting technical support is required. This is a reasonable step, but it does not address the possibility of uploading the library to other repositories.
- All accounts associated with domains that became available for sale were disabled. Their restoration also requires contacting technical support. This step is logical and useful, but it does not resolve the issue described in the previous point.
- Other comments from Sonatype related to using their products as a solution to the problem. However, many companies use their own or alternative solutions and do not see the need to switch to Sonatype products, making these comments more promotional.

Other repository owners did not provide comments on the situation. Sonatype made considerable efforts to improve the situation by monitoring problematic projects, which is commendable, but overall, the situation remains unchanged.

Conclusions from this situation are quite ambiguous. On the one hand, the described scenario represents a simple way to attack the software supply chain. The only technical difficulty is adding malicious code to the libraries. The problem is exacerbated by the fact that many components are used as transitive dependencies, and their application in the system often remains unnoticed. A vulnerable library can be used in one project, which is then integrated into another project, creating a chain reaction of vulnerabilities. Thus, under unfavorable circumstances, all projects can become vulnerable and susceptible to attack.

On the other hand, everything depends on the project build configuration: which repositories are used first and which ones second. Many projects may be protected from this attack due to the order of repository definition [9].

b) Updating libraries: Regularly updating dependencies to the latest stable versions that include vulnerability fixes.

c) Version control and dependency management: Implementing strict dependency management rules, including the use of declarative dependency files and lock files.

d) Security monitoring: Continuously monitoring vulnerabilities in used libraries with specialized services such as Snyk and WhiteSource.

e) Additionally, security reports can often be obtained from the GitHub security scanner or during the npm install, indicating security vulnerabilities in dependencies. These vulnerabilities rarely exist in the packages directly depended upon—they often exist in packages upon which dependencies themselves depend. Ideally, these vulnerabilities are immediately fixed by bots.

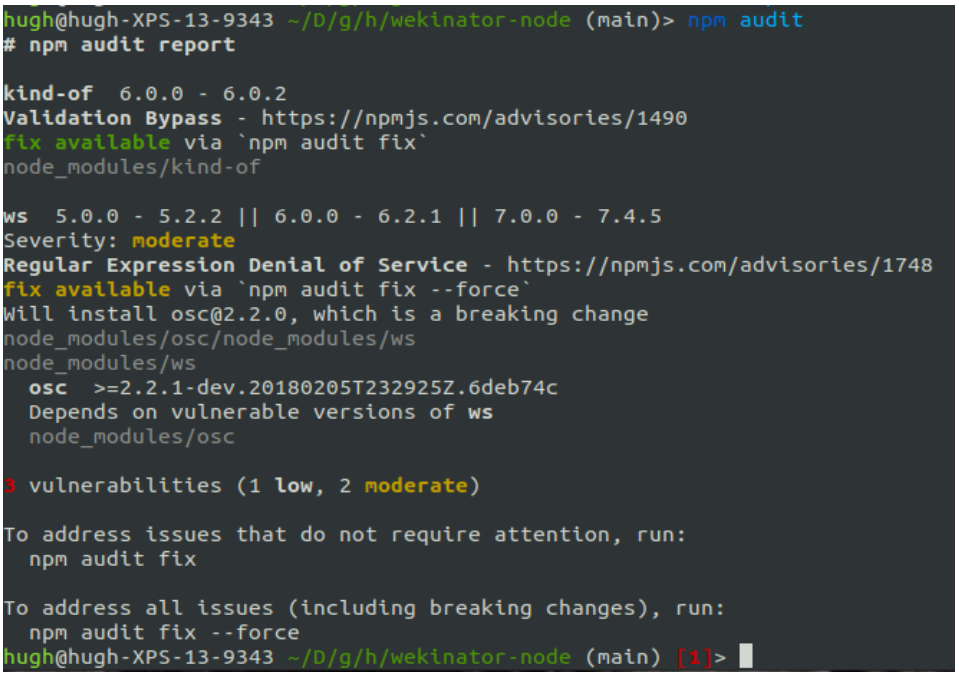

Figure 3: An Example of Vulnerability Correction [10].

In this case, a security vulnerability was flagged during the npm install, indicating that the vulnerability can be fully resolved by running the npm audit fix. This command recognizes that the version of the vulnerable package containing the fix is within the range specified by the dependent package. It can detect the vulnerable package in package-lock.json, and the issue will be resolved.

If you receive a warning from GitHub's Dependable security system, you can request a fix and submit a pull request. However, if the fixed version of the vulnerable package is not available, you will need to make a balanced decision. You can either submit pull requests for each required dependency in the chain and hope for a response from the maintainers or ignore the vulnerability. This involves balancing resources, and the optimal solution will depend on your context. However, it is important not to fall into the trap of ignoring one vulnerability and, consequently, all others. Upon identifying a new vulnerability, all deployments were paused to prevent it from being introduced into the production environment [10].

6. Practical part

Let's consider a simple example of a Java project that uses a third-party library, LibraryA, which in turn depends on LibraryB. Suppose a vulnerability is discovered in LibraryB.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.example</groupId>
    <artifactId>MyProject</artifactId>
    <version>1.0-SNAPSHOT</version>

    <dependencies>
        <!-- Direct dependency -->
        <dependency>
            <groupId>com.example</groupId>
            <artifactId>LibraryA</artifactId>
            <version>1.0.0</version>
        </dependency>
    </dependencies>
</project>
```

Fig.4. Using the library in the code

Transitive Dependency. LibraryA has the following dependency in its POM file:

```
<dependencies>
    <dependency>
        <groupId>com.example</groupId>
        <artifactId>LibraryB</artifactId>
        <version>1.0.0</version>
    </dependency>
</dependencies>
```

Figure 5: Transitive dependence

Suppose a vulnerability is found in LibraryB version 1.0.0 that allows an attacker to execute arbitrary code on the server (e.g., CVE-2023-12345). To fix the vulnerability, LibraryB needs to be updated to a secure version. This can be done by overriding the version of the transitive dependency in MyProject's POM file.

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.example</groupId>
    <artifactId>MyProject</artifactId>
    <version>1.0-SNAPSHOT</version>

    <dependencies>
        <!-- Direct dependency -->
        <dependency>
            <groupId>com.example</groupId>
            <artifactId>LibraryA</artifactId>
            <version>1.0.0</version>
        </dependency>

        <!-- Override transitive dependency version -->
        <dependency>
            <groupId>com.example</groupId>
            <artifactId>LibraryB</artifactId>
            <version>1.1.0</version> <!-- Secure version without the vulnerability -->
        </dependency>
    </dependencies>
</project>
```

Figure 6: Updated pom.xml

Fixing transitive vulnerabilities is crucial for maintaining the security of applications. In this example, a project that directly depends on LibraryA becomes vulnerable due to its transitive dependency on LibraryB. Updating transitive dependencies to secure versions helps prevent potential attacks and protects end users. Using automated tools for monitoring and analyzing dependencies, such as OWASP Dependency-Check, can significantly ease this process and ensure timely fixes for vulnerabilities.

Fixing transitive vulnerabilities in Java libraries can affect code efficiency in various ways, depending on the specific changes made in the updated libraries.

Let's consider a simple example where we measure the application's performance before and after updating the vulnerable library.

```java
import org.apache.commons.codec.digest.DigestUtils;

public class Main {
    public static void main(String[] args) {
        long startTime = System.nanoTime();

        for (int i = 0; i < 1000000; i++) {
            String hash = DigestUtils.sha256Hex("exampleString" + i);
        }

        long endTime = System.nanoTime();
        long duration = (endTime - startTime) / 1000000; // in milliseconds
        System.out.println("Duration: " + duration + " ms");
    }
}
```

Figure 7: An example of code that has a transitive vulnerability

After updating the library, additional security checks may have been introduced.

```java
import org.apache.commons.codec.digest.DigestUtils;

public class Main {
    public static void main(String[] args) {
        long startTime = System.nanoTime();

        for (int i = 0; i < 1000000; i++) {
            String hash = DigestUtils.sha256Hex("exampleString" + i);
        }

        long endTime = System.nanoTime();
        long duration = (endTime - startTime) / 1000000; // in milliseconds
        System.out.println("Duration: " + duration + " ms");
    }
}
```

Figure 8: Code overview after fixing transitive vulnerabilities

Thus, fixing transitive vulnerabilities in Java libraries can have both positive and negative impacts on code efficiency. Positive aspects include improved performance and stability due to optimizations and bug fixes. Ultimately, enhancing security and reducing the risk of exploiting vulnerabilities usually outweigh minor performance losses, resulting in a more reliable and secure application.

7. Conclusion

Managing vulnerabilities in dependencies is a critical task for ensuring the security and stability of your project. Understanding the differences between direct and transitive dependencies allows for more effective handling of security issues. Updating vulnerable packages to the latest patched versions is a key step in this process [11].

To minimize the risks associated with these vulnerabilities, a comprehensive approach is necessary, including automated dependency analysis tools, regular library updates, strict version control, and continuous security monitoring. Implementing these measures will significantly reduce the likelihood of vulnerability exploitation and enhance the overall security level of the software.

Using automated dependency analysis tools, such as Maven Dependency Plugin and OWASP Dependency-Check, and regularly updating libraries to the latest stable versions are key measures for preventing and fixing such vulnerabilities.

Strict version control and continuous security monitoring complement this approach, reducing risks and enhancing software protection. Adopting a comprehensive approach to dependency management will enable developers to create more reliable and secure applications, minimizing potential threats and ensuring high-quality end products.

References

[1] The state of the software supply chain. [Electronic resource] Access mode: https://www.sonatype.com/state-of-the-software-supply-chain/introduction (accessed 06.06.2024).
[2] Security in Java: best practices. [Electronic resource] Access mode: https://javarush.com/groups/posts/2713-bezopasnostjh-v-java-best-practices (accessed 06.06.2024).
[3] The Ultimate Guide To Effective Learning. [Electronic resource] Access mode: https://habr.com/ru/articles/816895 / (accessed 05/22/2024).
[4] Fixing vulnerabilities in Maven projects. [Electronic resource] Access mode: https://dev.to/brianverm/fixing-vulnerabilities-in-maven-projects-2686 (accessed 05/22/2024).
[5] Project quality indicators. [Electronic resource] Access mode: https://www.sonatype.com/resources/state-of-the-software-supply-chain-2022/project-quality-metrics (accessed 06.06.2024).
[6] Should security engineers take care of transit supply chain vulnerabilities? [Electronic resource] Access mode: https://semgrep.dev/blog/2023/transitive-supply-chain-vulnerabilities (accessed 06.06.2024).
[7] Using Maven for transit traffic. [Electronic resource] Access mode: https://codetinkering.com/transitive-cve-vulnerability-fixes / (accessed 06.06.2024).
[8] Fix the example of a transitive vulnerability for Maven. [Electronic resource] Access mode: https://docs .veracode.com/r/Fix_Example_Transitive_Vulnerability_for_Maven (accessed 06.06.2024).
[9] We are dealing with MavenGate, a new free installation for Java and Android applications. [Electronic resource] Access mode: https://habr.com/ru/companies/swordfish_security/articles/790544 / (accessed 06.06.2024).
[10] Fixing a vulnerability related to transient dependencies for npm. [Electronic resource] Access mode: https://www.hughrawlinson.me/posts/2021/06/21/transitive-dependency-vulnerability-resolution-for-npm (accessed 06.06.2024).
[11] Recommendations for eliminating vulnerabilities of transient dependencies. [Electronic resource] Access mode: https://www.thecodebuzz.com/fixing-transitive-dependency-vulnerabilities-best-practices / (accessed 06.06.2024).