Securing the software supply chain

Charles Zumbaugh
Dept. of Computer Science
Kansas State University
Manhattan, Kansas
cazumbaugh@ksu.edu

***Abstract*—Modern software development depends heavily on layers of abstraction such as libraries, frameworks, cloud infrastructure, build tooling, and other software built and maintained by others. This speeds innovation and makes the development of complex systems more accessible by allowing developers to build upon previous work. Given it's importance to modern development, the software supply chain represents an important attack vector. Attackers are increasingly targeting this ecosystem to mount attacks, and recent incidents highlight the scale of the fallout when these attacks are successful. This review discusses the primary attack vectors in the software supply chain, security measures and frameworks to defend against these attacks, and recent high-profile incidents.**

***Index Terms*—memory corruption, security, vulnerability, systems, malware**

I. Introduction

As computers become more embedded in everyday life, the security of these systems have become a critical concern. Cyber attacks against government and non-government organizations have increased in recent years, particularly against healthcare and large, multinational companies [1]. As personal data is increasingly being collected and stored on servers, the scale of these breaches is also increasing. For example, a recent breach of Change Healthcare in 2025 exposed the personal data of more than 192 million individuals in the United States [2]. The financial implications of security vulnerabilities are substantial, and IBM estimated the cost of a data breach in 2025 to be $4.4 million [3].

Modern software is built upon layers of reusable abstractions such as libraries, frameworks, cloud infrastructure, and build tools that are built and maintained by third parties. This ecosystem is known as the software supply chain and it involves numerous, globally-distributed participants building software used at various phases in the development process. The 2025 Open Source Security and Risk Analysis report found that 97% of evaluated codebases contained open source software, with 100% of codebases in the EdTech, internet, and mobile app sectors containing open source software [4].

Given the massive scale of open source software and the software supply chain in general, it is not surprising that attackers are increasingly targeting this ecosystem. Sonatype reported that 34,319 new open source malware packages were identified in quarter 3 of 2025, representing a 140% increase from quarter 2 2025 [5]. Recent attacks such as Log4j and SolarWinds highlight the importance of securing this ecosystem and the consequences of a successful attack. Aside from the economic damages associated with these attacks, distrust in the software ecosystem will undoubtedly slow technological innovation. Thus, it is essential to develop tools and frameworks to guard against malicious actors.

II. Software Supply Chain Attack vectors

Software supply chain attacks are considered to have three major attack vectors: dependencies, build infrastructure, and humans [6]. These attack vectors span the entire software lifecycle, and thus, security risks exist in each phase of this lifecycle. This review will focus primarily on software dependencies and artifact registries, cloud infrastructure, and build systems.

A. Dependencies

Vulnerabilities and malware included in open source and third party dependencies represent a critical security threat. In many cases, unintentional vulnerabilities are included in these dependencies. While such vulnerabilities are generally discovered, patched, and made public, developers and administrators may be slow to patch. For example, many systems were still vulnerable to attacks such as Heartbleed and Shellshock after the patch was released, in some cases years after [7]. A recent analysis of major package managers found that technical lag is common, with the majority of fixed version declarations being outdated and a significant number of flexible version declarations being outdated [8]. While there are likely many reasons for this, research has shown a strong presence of technical lag caused by the use of dependency constraints in the JavaScript package manager, npm, suggesting that developers may be reluctant to update dependencies due to backwards compatibility [9]. This idea is supported by qualitative research which demonstrated that developers are more likely to adopt a security fix *not* bundled with functionality improvements because they are less likely to introduce breaking changes [10]. Maintaining software ecosystems is costly, especially if updates are frequent [11], but the technical lag generated by missing updates can result in an increased risk of critical vulnerabilities.

Attackers may also attempt to infect programs by typosquatting, the practice of uploading a package with a similar name as that of a very popular package (ex. reacte vs react). If a developer makes a typo while installing the package (ex. *npm install reacte*), the malicious software is installed, which can target either the developer's or end user's machine. This

practice is common in software libraries and other areas of the internet. While the effectiveness at tricking users varies based on how the fake name is composed. Users are more likely to correctly identify typosquatting that adds characters to the name, and are more likely to be deceived by permutations and substitutions of characters [12]. Due to its prevalence, several software tools have been developed in recent years to guard against these attacks. A recently developed tool for identifying and reporting typosquatted imports, TypoGard, was able to flag up to 99.4% of known typosquatted imports on npm, PyPI, and RubyGems [13]. At a high level TypoGard compares the package name against a list of popular packages, and flags it if the name matches after transformations. These tools also exist to protect against typosquatted URLs online with good results. For example, TypoWriter uses a recurrent neural network to identify probable typo variations during the pre-registration phase [14], and TypoAlert is a Chrome extension that can alert users to alert users when they attempt to visit a typosquatted website [15].

B. Artifact Registries

An artifact registry or repository is a centralized system that stores, manages, and versions software artifacts such as libraries, compiled code, and container images. These registries provide a mechanism for package developers to deploy and maintain their software, and allow developers to easily consume these packages using package managers. Attacks on artifact registries are closely related to dependency vulnerabilities, as they commonly rely on developers unknowingly installing malicious software. Attackers attempt to hijack legitimate packages in these registries by attacking deleted packages, compromising the maintainer's account, or by exploiting package relocation in decentralized registries [16]. In the first case, attackers simply monitor the repositories package list for recently deleted packages, and publish a malicious version using the same name. Similarly, if an attacker can compromise the maintainer's account they can add malicious code in new versions or release malicious packages under the maintainer's name.

Exploiting package relocation is interesting in that it does not require an attacker to compromise the maintainer's account to succeed. This attack is described as a novel attack vector by Gu et al. (2023), and is briefly described below. Certain registries do not use a web portal to manage packages and instead rely on code hosting platforms such as GitHub for maintainers to publish their software. Instead, these registries maintain a list of packages and their corresponding URLs. These hosting platforms allow users to transfer ownership to another account, which will redirect the package from the old location to the new location. However, the registry is unaware of this change and will not automatically update the location of the package. Thus, if the original account is deleted an attacker can re-register the account and cut off the automatic GitHub redirection. This results in users installing a malicious library when they attempt to install or upgrade the package from the official registry.

C. Source Code Repositories

While artifact registries allow developers to deploy and manage software, source code repositories allow them to maintain and version the source code associated with that software. Use of these repositories is ubiquitous, allowing teams of developers to collaborate on all aspects of software projects, both closed- and open-source. More recently, automation pipelines such as GitHub Actions were integrated into these repositories and allow developers to automate processes such as testing, merging, and build. These pipelines allow users to configure various jobs to be run each time an event occurs, such as when code is pushed to the repository, and issue is closed, or a pull request is submitted. However, these workflows are not immune to vulnerabilities and represent a significant attack surface.

Reusable actions allow developers to create a workflow and call it from other workflows in the same project, organization, or across GitHub. For example, GitHub allows developers to publish their actions on GitHub Marketplace which can then be used by others in a similar manner as software dependencies. These actions can be malicious in nature, or contain security vulnerabilities unintentionally included by their creators. A recent review of reusable actions indicated that 54% were affected by at least one security weakness, with seven out of the top ten vulnerabilities associated with improper input validation [17].

Since actions allow for program execution, they can be affected by vulnerabilities in dependencies. JavaScript Actions are the most common type of reusable action [17], which allow JavaScript to be executed in a Node.js environment. Many of these actions rely on npm packages, and a recent analysis indicated that many of these dependencies are deeply nested, with 91% having indirect dependencies [18]. Deep nesting obfuscates understanding of the imported software, and this can lead to critical vulnerabilities being inadvertently included in these workflows.

Aside from vulnerabilities in the workflows associated with these repositories, security vulnerabilities can arise from contributions from bad actors, especially in the context of open-source.

D. Cloud Infrastructure

Cloud computing has boomed in recent years, with companies such as Amazon, Google, Microsoft, and others providing cloud infrastructure to users and companies at an incredible scale. This service offers scalability, flexibility, and initial cost savings compared to traditional infrastructure, but brings new security challenges as an external party must be entrusted with the data. Given their role and data they have access to, it is no surprise that security is one of the primary concerns. Additionally, cloud infrastructure brings with it new security issues related to virtualization, data geolocation, storage, confidentiality, and session hijacking [19]. At a high level, the security of cloud infrastructure can be broken into four levels: the compute level (ex. physical server and hypervisor security), the network level (ex. virtual firewall and securing

data in transit), the application level (ex. providing protection to applications utilizing the resources), and data [20].

At the data level, cloud infrastructure faces issues such as data breaches, loss, segregation, virtualization, confidentiality, integrity, and availability [21]. The shared nature of cloud infrastructure makes it a target for attackers, as success can mean stealing a significant amount of data. While there are many reasons for data loss and leakage, the primary reasons are due to misconfigurations, unauthorized access, insecure interfaces, and hijacking accounts [22]. Attacks can be carried out by internal or external threat actors, and encryption and watermarking are the two major defenses to these security issues [21].

Outside of data loss/leakage, cloud infrastructure providers face issues related to availability and integrity. Customers expect near constant uptime without data corruption. The distributed nature of cloud computing complicates these, and attackers can target availability by launching a distributed denial of service (DDoS) attack, and can attack integrity by corrupting any of the multiple locations the data is stored.

At the compute level, cloud infrastructure faces threats related to physical security and virtualization. To provide cloud infrastructure at a large scale, providers build or rent large data centers that provide the physical servers, storage, and networking. As these servers contain sensitive data, unauthorized access by employees or other individuals presents a major security threat. While physically accessing the server is infeasible for the vast majority of cybercriminals, they may gain access to the servers through vulnerabilities related to virtualization. Cloud computing heavily relies on virtualization, where many virtual machines share the same physical hardware. Vulnerabilities in the hypervisor can allow attackers to gain access to the physical hardware and the applications hosted by the hypervisor [21].

Cloud infrastructure is vulnerable to attacks on its network, much like other internet-based services and applications. Attackers can target cloud infrastructure using DDoS and domain name server (DNS) cache poisoning attacks, and packet sniffing. Additionally, IP addresses are commonly reassigned in cloud environments, which can increase the risk of attack if the process of reusing an IP address happens faster than its old assignment is removed from the DNS cache [21].

Finally, cloud infrastructure faces threats at the application level such as insecure APIs and end user attacks [21]. Cloud computing relies on numerous APIs that connect hardware and various software components such as databases, authentication and authorization systems, compute engines, and others. Vulnerabilities in these APIs can provide attackers a mechanism to carry out attacks. Additionally, attackers may target the end users of the services in phishing schemes to attempt to gain access to their accounts.

III. Defending against attacks

It is clear that there are many attack surfaces in the software supply chain, and the number of disparate components and organizations involved can make security difficult. Modern software applications are complex and commonly involve hundreds or thousands of dependencies, from software libraries to cloud infrastructure. An attacker requires just a single vulnerable component to mount an attack, and the result can affect businesses, end users, and governments. Software supply chain security has evolved in recent decades, evolving from a focus on perimeter defense, to including dependency management, software bill of materials (SBOM), zero-trust and secure build pipelines, and global standards [23]. Additionally, new tooling has been developed to assist organizations in implementing these methods.

A. Dependency Management

As software has become more complex, managing the hundreds or thousands of dependencies has become increasingly important. At a basic level, most package managers such as npm or Maven use versioning systems that provide users with the ability to specify automatic updates. For example, users can use the "$\wedge$" operator to allow for minor or patch updates, or the "$\sim$" to allow for only patch updates. However, management of dependencies becomes much more complex when transitive dependencies are considered. On average, a user implicitly includes around 80 additional packages through transitive dependencies for each package installed by npm, and up to 40% of packages rely on code known to be vulnerable [24]. Clearly, this is a security threat to modern applications. Zimmermann et al. (2019) present some potential mitigations to this including raising developer awareness, vulnerable package warnings, code vetting, and training and vetting maintainers. Most modern package managers provide warnings for vulnerable packages, but the vulnerability must be known and they generally do not address vulnerabilities in transitive dependencies. While manual code vetting is likely infeasible at scale, automated analyzers could be adopted by package registries to analyze each update [24]. Similar processes are already in use in other areas. For example, the Apple App Store performs static analysis on all submitted binaries to help maintain the security and privacy of the Apple ecosystem.

B. Software Bill of Materials

A SBOM is a formal record containing the details and supply chain relationships of various components used in building software [25]. The National Telecommunications and Information Administration (NTIA) provides some minimum elements for a SBOM, shown in Table I.

The SBOM is intended to provide developers and users of software with information regarding the supply chain, allowing them to track known and recently discovered security vulnerabilities. Recent high profile supply chain cyber attacks, such as Log4Shell, have highlighted the need for SBOMs, as they allowed companies to very quickly patch the vulnerability [26]. There are three primary formats for SBOMs: 1) Software Package Data eXchange (SPDX), 2) CycloneDX, and 3) Software Identification (SWID) Tagging [27]. In 2021, the

TABLE I: Minimum elements of a SBOM as defined by NTIA [25]

Item
Supplier
Component name
Version of component
Unique identifiers
Dependency relationship
Author of SBOM
Timestamp
Provide support for SBOM automation
Define the operations of SBOM requests, generation, and use

White House issues an executive order requiring all companies conducting business with the US government to provide SBOMs [28]. However, despite these mandates, adoption in the industry and open source community is lagging. A large proportion of widely used software do not have SBOMs, and there is a lack of consensus regarding what to include in SBOMs, despite the official NTIA recommendations [27]. The lack of widespread adoption is likely due, in part, to challenges with automated SBOM tools, lack of interoperability, time consumption, and risks of transparency [29].

C. Zero-trust build pipelines

D. Units

- Use either SI (MKS) or CGS as primary units. (SI units are encouraged.) English units may be used as secondary units (in parentheses). An exception would be the use of English units as identifiers in trade, such as "3.5-inch disk drive".
- Avoid combining SI and CGS units, such as current in amperes and magnetic field in oersteds. This often leads to confusion because equations do not balance dimensionally. If you must use mixed units, clearly state the units for each quantity that you use in an equation.
- Do not mix complete spellings and abbreviations of units: "Wb/m^2" or "webers per square meter", not "webers/m^2". Spell out units when they appear in text: ". . . a few henries", not ". . . a few H".
- Use a zero before decimal points: "0.25", not ".25". Use "cm^3", not "cc".)

E. Equations

Number equations consecutively. To make your equations more compact, you may use the solidus (/), the exp function, or appropriate exponents. Italicize Roman symbols for quantities and variables, but not Greek symbols. Use a long dash rather than a hyphen for a minus sign. Punctuate equations with commas or periods when they are part of a sentence, as in:

$$a + b = \gamma \tag{1}$$

Be sure that the symbols in your equation have been defined before or immediately following the equation. Use "(1)", not "Eq. (1)" or "equation (1)", except at the beginning of a sentence: "Equation (1) is . . ."

F. LaTeX-Specific Advice

Please use "soft" (e.g., `\eqref{Eq}`) cross references instead of "hard" references (e.g., `(1)`). That will make it possible to combine sections, add equations, or change the order of figures or citations without having to go through the file line by line.

Please don't use the `{eqnarray}` equation environment. Use `{align}` or `{IEEEeqnarray}` instead. The `{eqnarray}` environment leaves unsightly spaces around relation symbols.

Please note that the `{subequations}` environment in LaTeX will increment the main equation counter even when there are no equation numbers displayed. If you forget that, you might write an article in which the equation numbers skip from (17) to (20), causing the copy editors to wonder if you've discovered a new method of counting.

BibTeX does not work by magic. It doesn't get the bibliographic data from thin air but from .bib files. If you use BibTeX to produce a bibliography you must send the .bib files.

LaTeX can't read your mind. If you assign the same label to a subsubsection and a table, you might find that Table I has been cross referenced as Table IV-B3.

LaTeX does not have precognitive abilities. If you put a `\label` command before the command that updates the counter it's supposed to be using, the label will pick up the last counter to be cross referenced instead. In particular, a `\label` command should not go before the caption of a figure or a table.

Do not use `\nonumber` inside the `{array}` environment. It will not stop equation numbers inside `{array}` (there won't be any anyway) and it might stop a wanted equation number in the surrounding equation.

G. Some Common Mistakes

- The word "data" is plural, not singular.
- The subscript for the permeability of vacuum μ_0, and other common scientific constants, is zero with subscript formatting, not a lowercase letter "o".
- In American English, commas, semicolons, periods, question and exclamation marks are located within quotation marks only when a complete thought or name is cited, such as a title or full quotation. When quotation marks are used, instead of a bold or italic typeface, to highlight a word or phrase, punctuation should appear outside of the quotation marks. A parenthetical phrase or statement at the end of a sentence is punctuated outside of the closing parenthesis (like this). (A parenthetical sentence is punctuated within the parentheses.)
- A graph within a graph is an "inset", not an "insert". The word alternatively is preferred to the word "alternately" (unless you really mean something that alternates).
- Do not use the word "essentially" to mean "approximately" or "effectively".
- In your paper title, if the words "that uses" can accurately replace the word "using", capitalize the "u"; if not, keep using lower-cased.

- Be aware of the different meanings of the homophones "affect" and "effect", "complement" and "compliment", "discreet" and "discrete", "principal" and "principle".
- Do not confuse "imply" and "infer".
- The prefix "non" is not a word; it should be joined to the word it modifies, usually without a hyphen.
- There is no period after the "et" in the Latin abbreviation "et al.".
- The abbreviation "i.e." means "that is", and the abbreviation "e.g." means "for example".

An excellent style manual for science writers is.

H. Authors and Affiliations

The class file is designed for, but not limited to, six authors. A minimum of one author is required for all conference articles. Author names should be listed starting from left to right and then moving down to the next line. This is the author sequence that will be used in future citations and by indexing services. Names should not be listed in columns nor group by affiliation. Please keep your affiliations as succinct as possible (for example, do not differentiate among departments of the same organization).

I. Identify the Headings

Headings, or heads, are organizational devices that guide the reader through your paper. There are two types: component heads and text heads.

Component heads identify the different components of your paper and are not topically subordinate to each other. Examples include Acknowledgments and References and, for these, the correct style to use is "Heading 5". Use "figure caption" for your Figure captions, and "table head" for your table title. Run-in heads, such as "Abstract", will require you to apply a style (in this case, italic) in addition to the style provided by the drop down menu to differentiate the head from the text.

Text heads organize the topics on a relational, hierarchical basis. For example, the paper title is the primary text head because all subsequent material relates and elaborates on this one topic. If there are two or more sub-topics, the next level head (uppercase Roman numerals) should be used and, conversely, if there are not at least two sub-topics, then no subheads should be introduced.

J. Figures and Tables

a) Positioning Figures and Tables: Place figures and tables at the top and bottom of columns. Avoid placing them in the middle of columns. Large figures and tables may span across both columns. Figure captions should be below the figures; table heads should appear above the tables. Insert figures and tables after they are cited in the text. Use the abbreviation "Fig. 1", even at the beginning of a sentence.

Figure Labels: Use 8 point Times New Roman for Figure labels. Use words rather than symbols or abbreviations when writing Figure axis labels to avoid confusing the reader. As an example, write the quantity "Magnetization", or "Magnetization, M", not just "M". If including units in the label, present them within parentheses. Do not label axes only with units. In the example, write "Magnetization (A/m)" or "Magnetization {A[m(1)]}", not just "A/m". Do not label axes with a ratio of quantities and units. For example, write "Temperature (K)", not "Temperature/K".

TABLE II: Table Type Styles

Table Head	Table Column Head		
	Table column subhead	*Subhead*	*Subhead*
copy	More table copy[a]		

[a]Sample of a Table footnote.

Fig. 1: Example of a figure caption.

ACKNOWLEDGMENT

The preferred spelling of the word "acknowledgment" in America is without an "e" after the "g". Avoid the stilted expression "one of us (R. B. G.) thanks ...". Instead, try "R. B. G. thanks...". Put sponsor acknowledgments in the unnumbered footnote on the first page.

REFERENCES

tnotes in the abstract or reference list. Use letters for table footnotes.

REFERENCES

[1] H. Hammouchi, O. Cherqi, G. Mezzour, M. Ghogho, and M. El Koutbi, "Digging deeper into data breaches: An exploratory data analysis of hacking breaches over time," *Procedia Computer Science*, vol. 151, pp. 1004–1009, 2019.

[2] P. R. Clearinghouse. (2025) Data breaches. [Online]. Available: https://privacyrights.org/data-breaches

[3] IBM. (2025) Cost of a data breach report 2025. [Online]. Available: https://www.ibm.com/reports/data-breach

[4] B. Duck, "2025 open source security and risk analysis report," Black Duck, Tech. Rep., 2025.

[5] S. S. R. Team. (2025) Open source malware index q3 2025: High-severity attacks surge. [Online]. Available: https://www.sonatype.com/blog/open-source-malware-index-q3-2025

[6] L. Williams, G. Benedetti, S. Hamer, R. Paramitha, I. Rahman, M. Tamanna, G. Tystahl, N. Zahan, P. Morrison, Y. Acar *et al.*, "Research directions in software supply chain security," *ACM Transactions on Software Engineering and Methodology*, vol. 34, no. 5, pp. 1–38, 2025.

[7] B. Hammi and S. Zeadally, "Software supply-chain security: Issues and countermeasures," *Computer*, vol. 56, no. 7, pp. 54–66, 2023.

[8] J. Stringer, A. Tahir, K. Blincoe, and J. Dietrich, "Technical lag of dependencies in major package managers," in *2020 27th Asia-Pacific Software Engineering Conference (APSEC)*, 2020, pp. 228–237.

[9] A. Zerouali, E. Constantinou, T. Mens, G. Robles, and J. González-Barahona, "An empirical analysis of technical lag in npm package dependencies," in *International conference on software reuse*. Springer, 2018, pp. 95–110.

[10] I. Pashchenko, D.-L. Vu, and F. Massacci, "A qualitative study of dependency management and its security implications," in *Proceedings of the 2020 ACM SIGSAC conference on computer and communications security*, 2020, pp. 1513–1531.
[11] S. Berhe, M. Maynard, and F. Khomh, "Maintenance cost of software ecosystem updates," *Procedia Computer Science*, vol. 220, pp. 608–615, 2023.
[12] J. Spaulding, D. Nyang, and A. Mohaisen, "Understanding the effectiveness of typosquatting techniques," in *Proceedings of the fifth ACM/IEEE Workshop on Hot Topics in Web Systems and Technologies*, 2017, pp. 1–8.
[13] M. Taylor, R. Vaidya, D. Davidson, L. De Carli, and V. Rastogi, "Defending against package typosquatting," in *International conference on network and system security*. Springer, 2020, pp. 112–131.
[14] I. Ahmad, M. A. Parvez, and A. Iqbal, "Typowriter: a tool to prevent typosquatting," in *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*, vol. 1. IEEE, 2019, pp. 423–432.
[15] F. Blefari, A. Furfaro, G. Ianni, A. Viscomi *et al.*, "Typoalert: a browser extension against typosquatting," in *Proc. of SEBD: 32nd Symposium on Advanced Database Systems*, 2024.
[16] Y. Gu, L. Ying, Y. Pu, X. Hu, H. Chai, R. Wang, X. Gao, and H. Duan, "Investigating package related security threats in software registries," in *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2023, pp. 1578–1595.
[17] H. O. Delicheh and T. Mens, "Mitigating security issues in github actions," in *Proceedings of the 2024 ACM/IEEE 4th International Workshop on Engineering and Cybersecurity of Critical Systems (EnCyCriS) and 2024 IEEE/ACM Second International Workshop on Software Vulnerability*, 2024, pp. 6–11.
[18] H. Onsori Delicheh, A. Decan, and T. Mens, "Quantifying security issues in reusable javascript actions in github workflows," in *Proceedings of the 21st International Conference on Mining Software Repositories*, 2024, pp. 692–703.
[19] A. Djenna and M. Batouche, "Security problems in cloud infrastructure," in *The 2014 International Symposium on Networks, Computers and Communications*. IEEE, 2014, pp. 1–6.
[20] H. Saini and A. Saini, "Security mechanisms at different levels in cloud infrastructure," *International Journal of Computer Applications*, vol. 108, no. 2, pp. 1–6, 2014.
[21] Y. Alghofaili, A. Albattah, N. Alrajeh, M. A. Rassam, and B. A. S. Al-Rimy, "Secure cloud infrastructure: A survey on issues, current solutions, and open challenges," *Applied Sciences*, vol. 11, no. 19, p. 9005, 2021.
[22] B. M. Salih and O. K. J. Mohammad, "Cloud data leakage, security, privacy issues and challenges," *Procedia Computer Science*, vol. 242, pp. 592–601, 2024.
[23] S. Anasuri and G. P. Rusum, "Software supply chain security: Policy, tooling, and real-world incidents," *International Journal of Emerging Trends in Computer Science and Information Technology*, vol. 5, no. 3, pp. 79–89, 2024.
[24] M. Zimmermann, C.-A. Staicu, C. Tenny, and M. Pradel, "Small world with high risks: A study of security threats in the npm ecosystem," in *28th USENIX Security symposium (USENIX security 19)*, 2019, pp. 995–1010.
[25] NTIA, "The minimum elements for a software bill of materials (sbom)," *https://www. ntia. doc. gov/files/ntia/publications/sbom_minimum_elements_report. pdf*, 2021.
[26] P. Roberts. (2021) Log4j: Why your organization needs to embrace software bills of materials. [Online]. Available: https://www.reversinglabs.com/blog/log4j-is-why-you-need-an-sbom
[27] B. Xia, T. Bi, Z. Xing, Q. Lu, and L. Zhu, "An empirical study on software bill of materials: Where we stand and the road ahead," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 2630–2642.
[28] W. House, "Executive order on improving the nation's cybersecurity," *The White House: Presidential Actions*, 2021.
[29] N. Zahan, E. Lin, M. Tamanna, W. Enck, and L. Williams, "Software bills of materials are required. are we there yet?" *IEEE Security & Privacy*, vol. 21, no. 2, pp. 82–88, 2023.