International Journal of Core Engineering & Management
Volume-7, Issue-06, 2023 ISSN No: 2348-9510

MITIGATING SUPPLY CHAIN ATTACKS IN WEB APPLICATIONS: A CASE STUDY ON LOG4J AND SPRING4SHELL

Sandeep Phanireddy
USA
phanireddysandeep@gmail.com

Abstract

Recently supply chain attacks in web applications are becoming a typical cybersecurity threat in which attackers take advantage of vulnerabilities of widely used software components. The attacks focus on third party dependencies, package repos as well as software update mechanisms to compromise entire systems. The Log4j (CVE-2021-44228) and Spring4Shell (CVE-2022-22965) vulnerabilities exemplify the destructive scope vulnerabilities in the most popular frameworks harbor, either leading to a complete compromise of service, data leakage, and the complete execution involving the remote code execution (RCE). These are indicative of a necessity for robust dependency management and pro activeness in security where a web application is concerned. Various detection techniques and mitigation strategies are explored which meet the supply chain threats in this paper. The main methodologies used are Software Bill of Material (SBOM) for monitoring dependencies; proxy repositories to stop unapproved package downloads; and dependency scanning tools such as, OWASP Dependency-Check, Snyk and GitHub Dependabot. The paper also notes about the necessity of static and dynamic analysis (SAST, DAST) and regular patch management to reduce exploitation risk. Integrating these strategies increases the ability of organizations to harden their security posture to modern supply chain threats. By proposing an exhaustive framework to identify and mitigate prevent Software Supply Chain attacks, this work extends the thrust of literature by emphasizing on the fact that future should see continuous monitoring and secure development practice along with AI based threat detection.

Keywords: Supply Chain Attacks, Dependency Management, Remote Code Execution (RCE), Software Bill of Materials (SBOM), Vulnerability Mitigation

I. INTRODUCTION

1.1 Understanding Supply Chain Attacks in Web Applications

Today, most modern web applications are derived through a combination of proprietary code, open-source libraries, as well as 3rd party components [1]. This brings much faster development and innovation, but also fierce security risks. Supply chains attacks are one of the most critical threats borne by organizations today, where the malicious actors try to seize control of the software vulnerabilities as it goes through various phases of development and deployment processes. Supply chain attacks are different from the regular cyberattacks through the exploitation of weakness inside organization's internal systems by attacking 3rd party components, which can make it tough to discover and eliminate the effects. In recent years

however, these attacks have become much more popular thanks to the boom of OSS [2]. This is particularly true when an organization integrates open source libraries in their program without performing thorough security evaluations, taking the assumption that these components are secure based on the fact that they are massively popular [3]. However, this trust is abused by attackers who inject malicious code into popular libraries, compromising package repositories or utilizing outdated dependencies with known vulnerabilities.

There are various ways in which these attacks could occur, i.e. injecting malicious code into popular open-source libraries, exploits in popular used frameworks, compromising package repos like Maven, for Java; npm, for JavaScript; and PyPI, for Python. A single vulnerable dependency can destabilize the entire software supply chain at once, spreading through thousands of organizations [4]. While attackers can do typo squatting — where they upload malicious packaging files with names close to legitimate ones or even take over abandoned or unmaintained packaging files to distribute malware during an [?]. One example of this is the Solar Winds attack in 2020, where compromised software update could be used to dig into government agencies and major corporations. There have also been similar incidents as in 2018 where an attacker was able to inject backdoors into a widely used JavaScript package and thus also influenced affected a large number of projects that depend on it. These external components are very main to web applications, and any security flaw in them can have incredible impacts, taking into account a great many downstream applications and their [?]. In order for organizations to meet these challenges, they are required to take active steps in curtailing reactive security paradigms through continuous monitoring of dependencies, as well as strict access controls and verification of integrity of external code before integrating it with software.

1.2 The Importance of Third-Party Dependencies

Nowadays, open-source software (OSS) is an essential part of modern development ecosystem [5]. Having less than 100 line of code and costing just 1 dollar per month it is economical, powerful and easy to work with. Instead of reinventing everything, developers can use existing prebuilt libraries to save time and accelerate development cycle and performance of the software [6]. This is made possible via package managers such as Maven (Java), npm (JavaScript) and PyPI (Python) for download, update and dependency management. While this reliance on third party code brings with it risks, such as security risks from malicious actors exploiting vulnerabilities in third-party code or injecting malicious code into software products via the maliciously modified package repository. There are a number or reasons why third party dependencies can be risky, namely insufficient visibility, inconsistently security practices and assumptions of trust [7]. Tracking which libraries and versions a given organization is using is difficult, which make it hard to respond to security advisory as quickly as one would like. However, attackers can also exploit known vulnerabilities if some developers don't update dependencies as it would lead to breaking of the existing functionality if it is used to update the dependencies [8]. Take a look at how supply chain risks have affected the different package ecosystems:

- Maven (Java): It is known that Java applications heavily depend on libraries, which are managed by Maven Central Repository and have thousands of open-source components. There are critical security breaches due to vulnerabilities in widely used Java libraries, such as Apache Struts, Jack- son, and Log4j. An enterprise that depends on Java for its software is

vulnerable to cyberattacks if a commonly used Java library is compromised: Thousands of enterprises from financial institutions to government agencies can be at risk.

- npm (JavaScript): Due to its extensive used in web applications, the npm ecosystem of JavaScript is one of the largest and most targeted. They have used typo squatting, dependency hijacking, and malicious updates to inject harmful code into the widely used npm packages. For example, the event stream attack in which compromised package was used by an attacker to steal cryptocurrency from applications which used the library.

- PyPI (Python): Attackers have also targeted Python's PyPI repository and distributed fake or compromised libraries there. It is often malicious actors that upload fake libraries with names close to their true name (e.g. requests instead of requests) that developers accidentally install the malware. Also, Python package maintainers simply fall off the map and leave their projects to the hands of attackers who are able to upload malicious updates.

Without security, when a third party is compromised the attacker can inject malware into the application, giving them remote access or data exfiltration, stealing sensitive information, etc. and gaining total application compromise. The supply chain attacks have grown 742% in the last 3 years, and as such organizations need to adopt the proactive security measures of Software Bill of Materials (SBOM), so they can always have an inventory of dependencies; automated security scanning tools such as OWASP Dependency Check and Sonatype Nexus, to limit exposure to any malicious packages using proxy repositories and enforcing strict Dependency Management policies preventing any unauthorized updates. With the rapid adoption of open-source components, software supply chain security has become top priority for developers, security teams and policymakers alike.

1.3 Notable Supply Chain Vulnerabilities: Log4j and Spring4Shell

Millions of applications worldwide were left vulnerable to cyber threats as supply chain attacks became a devastating force to be reckoned with, with the likes of Log4j(Log4Shell) and Spring4Shell. These incidents spotlight how a single weakness in a commonly utilized open supply library might one-day lead to disastrous outcomes within multiple industries, together with cloud providers, monetary establishments, wellbeing care programs and authorities' companies [9].

1.3.1 Log4j (CVE-2021-44228) - A Global Supply Chain Catastrophe

Log4j (CVE-2021–44228) was one of the largest and poorest supply chain attacks in history. In December 2021, Apache Log4j discovered this zero-day flaw in the Apache Log4j logging library, which resides within millions of Java based applications. Remote Code Execution (RCE) was possible, and all that was required for an attacker to take full control of affected systems was to send a specially crafted log message. That in turn meant that the flaw could be exploited even from untrusted user inputs, such as text typed in chat messages or HTTP headers [10]. That impact was real in the real world. As a result, major cloud providers such as Amazon AWS, Microsoft Azure and Google Cloud all had to get rapid in their patching of services. Less than days after the vulnerability was publicly announced, Cybercriminals and nation state of actors launched millions

of attacks against organizations such as Tesla, Apple and government agencies. It also was used to load malware and steal sensitive data by ransomware gangs [11].

The lesson learned from this incident was very critical in the area of dependency management and security [12]. Few organizations realized that they were using Log4j as they did not have a Software Bill of Materials (SBOM) in place to track. The exposure was extended because some companies did not have time to patch their dependencies, and as a result, stuck vulnerable [13]. The Log4j crisis ultimately underscored that it's time for dependency monitoring and security policy for 3rd party components to be proactive.

1.3.2 Spring4Shell (CVE-2022-22965) - Another Major Java Exploit

Spring4Shell had a big impact, affecting thousands of enterprise Java applications that are used for banking, e-commerce, and government systems. Automated exploits were developed by cybercriminals quickly to scan the internet for vulnerable systems who then had to update their Spring dependencies quickly to remain safe [14]. Spring4Shell affected thousands of enterprise Java applications used in banking, e-commerce and government systems on a wide scale. Cybercriminals very quickly developed automated exploits of this nature, and they began to quickly scan the Internet, searching for vulnerable systems, which required organizations to themselves update their Spring dependency packages very rapidly [15].

Key takeaways from the Spring4Shell incident noted the value of proactive patching to other organizations that aren't able to detect and remediate Spring4Shell through dependency monitoring. Besides, due to the large number of people affected when a framework uses a widely known vulnerability, security teams had to be more sensitive to the risks at the framework level. Moreover, secure software development practices (SSDLC) were required to mitigate the risks of such vulnerabilities in the future [16].

1.3.3 Key Security Takeaways from Log4Shell and Spring4Shell

The whole series of incidents revealed the holes in dependency management and software supply chain that prompted organizations to revise their security strategies [17]. Then, the CISA, NIST and OWASP responded by issuing new guidelines on how to improve the security of the software supply chain. The mandatory use of a Software Bill of Materials (SBOM) to track third party dependencies as well to give better visibility in the components that are in the use for software applications was one of the key recommendations. Foreseen was the implementation of automated vulnerability scanning for detecting such flaws in open source libraries before they can be exploited [18]. Other organizations were also advised to resort to proxy repositories such as Maven Nexus or Artifactory to prevent external dependencies from getting included without restricting their access [19]. Last, runtime protection schemes were suggested as a way to prevent exploits from being executed [20].

Log4Shell and Spring4Shell served as a reminder that we rely on continuous monitoring, rapid patching and secure software supply chain management. The risk theory of threat poses a threat of spreading the cyber threat to all kinds of organizations and relying on the correct implementation will no longer become possible due to the ever-evolving development of software.

International Journal of Core Engineering & Management
Volume-7, Issue-06, 2023 ISSN No: 2348-9510

1.4 Research Objective

This paper examines attacks in the real world, methods to detect them and best practices to reduce the risk. Specifically, it examines:

- Attackers' use of third-party dependencies in web applications.
- Where tools such as OWASP Dependency-Check, Sonatype Intelligence, and CISA SBOM come into play in identifying vulnerabilities.
- How to secure the software supply chain using SBOM (Software Bill of Materials), proxy repositories and secure package management via Maven, npm and PyPI platforms.

Familiarizing yourself with these threats and coming up with effective defensive techniques will help organizations mitigate risks of supply chain attacks and strengthen software ecosystems.

II. SUPPLY CHAIN ATTACKS IN WEB APPLICATIONS

2.1 Understanding Supply Chain Attacks

Web applications get attacked by supply chain attacks when the attackers attack third party components, libraries, or development tools with which the application is dependent [21]. Unlike classical cyberattacks that exploit (vulnerabilities) in organization's own systems, such cyberattacks rely on the interconnections among the modern software development [22]. Hackers now have a way to inject malicious code into the widely used dependencies, compromising thousands of applications at one time thus affecting businesses, governments and end users.

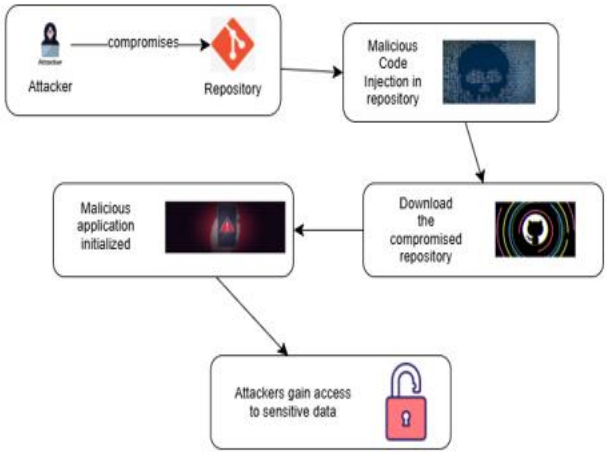

Figure 1: Supply chain attack

But now, with all these web applications using open source software and they rely heavily on libraries from the outside, their attack surface has just seen a big expansion [23]. At the developer level, we usually grab prebuilt components by using repository like Maven for Java, npm for JavaScript and PyPI for Python to speed up the development much faster. These package managers offer thousands of ready-to-use libraries, simplify coding, increase functionality, and save time in developing. But if any of these dependencies are broken down, attackers are given a stealth option to enter applications dependent upon them [24]. The reasons behind the numerous

supply chain attacks on web applications are:

2.1.1 Widespread Use of Open-Source Software

Instances of Fortune 500 companies using OSS are rampant and many organizations have embraced the use of this open source to a great extent [25]. Developers get free customizable and well tested solutions to accelerate the development when the projects are open source. We know that most OSS projects are monitored by volunteers or small teams with limited resources, which makes them prone to security gaps [26]. Widely used libraries could be left open to exploitation until security updates and patches catch up. This overreliance on OSS is the silver lining for the attacker: he can attack the libraries those organizations happen to be using that haven't kept their security up or that are infrequently updated. They can get into lot of applications by injecting the malicious code into the popular but placed poorly maintained repositories. Due to the fact that many of these dependencies are gauged in by organizations without much caution, even a single compromised package can have broad reach [27].

2.1.2 Complexity of Modern Software Development

With the growth in the modern web application, it has become a complex network of many dependencies. Third party libraries on your application end up dependent on each other and on a single application can have dozens if not hundreds of these libraries [28]. Because of this nested dependency structure, an entire ecosystem of applications could be compromised if one package in the supply chain has its integrity compromised. Dependency blindness does not mean that developers are unaware of the number of dependencies their applications have present [29]. The package that sits deep in a dependency chain is a good instance in which an attacker gains control of production software, and organizations might not even know it. Given that this creates an increasingly difficult problem of securing the software supply chain, increased visibility is needed.

2.1.3 Automated Package Management & Continuous Integration (CI/CD) Pipelines

Because of this, it is common for developers to use automated package managers like npm, PyPI, and Maven to fetch the latest dependency versions, and this is done in order to streamline development. They help with speeding this development and every application must access the latest version of features and bug fix [30]. Yet, this automation brings a security risk to updating itself if verification is not complete. As attackers exploit this automation, they inject malicious updates into legitimate libraries in order to pass undetected. The malicious code spreads across multiple projects in case of an unknowing developer that downloads and inserts a compromised package [8]. In high profile attacks, this tactic has been used including typo squatting (creating package names that are similar to existing popular locations) and dependency confusion (publishing fake versions of internal company packages to public repositories).

2.1.4 Difficulty in Detecting Malicious Code

Supply chain attacks are unlike many forms of malware, which sometimes can be caught with antivirus software, as they are done in such a stealthy and subtle manner that they result in changes to existing code [31]. They could be as crude as adding a few lines of code that exfiltrate sensitive data or that bring in a hidden backdoor. The reason is that the modifications are inside legitimate libraries, and as a result, they often bypass traditional security scans and code reviews.

International Journal of Core Engineering & Management
Volume-7, Issue-06, 2023 ISSN No: 2348-9510

Also, malicious code can stay dormant for months or even years before it activates meaning there is little of an indication to alert you [32]. The malicious payload can wait for a specific condition of an event, for example a software update or an expiration date. Such delay further complicates forensic investigations as it becomes difficult to locate the reason why an attack was not detected early or solved, as this occurred much later than the initial compromise.

2.2 Real-World Supply Chain Attacks

Supply chain attacks have very much been around for some time, as demonstrated by a number of high profile incidents, such as Log4Shell, Codecov and Solar Winds.

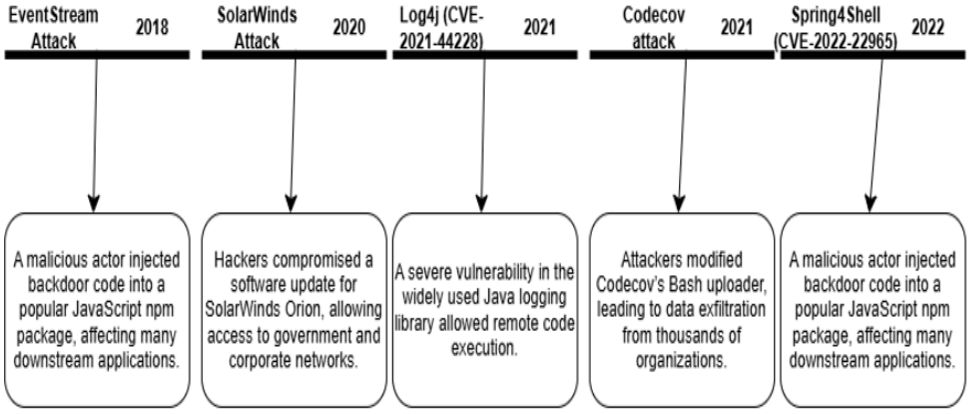

Figure 2: Timeline of Major Supply Chain Attacks

2.2.1 Log4Shell (Log4j Vulnerability - 2021)

The most important vulnerability of recent years, Log4Shell, was found in Apache Log4j, a popular Java logging framework. Attackers could remotely execute their own arbitrary code by sending a specially crafted request. It was an easy to exploit security threat that affected millions of applications including enterprise software and cloud services [33]. As a result, industries relied heavily on Log4j but with widespread use, major organizations such as Amazon, Microsoft, and Google were left scrambling to patch it to prevent possible breach. This was an event that showed just how big of a ripple effect a single open source vulnerability can have on the world software ecosystem at large [8].

2.2.2 SolarWinds Attack (2020)

This is the SolarWind's attack that showed how even the most secure organizations can be infiltrated through a supply chain compromise. Attackers inserted a backdoor (Sunburst malware) into routine software update of a popular network monitoring software SolarWinds Orion, so they could infiltrate the networks of the companies they targeted. After the update was installed, the attack granted the attackers the ability to access unauthorized sensitive networks. Unknown to customers over 18,000 of them had been deploying the undisclosed update [34]. It demonstrated just how dangerous it is to trust third party software update packages without getting your hands dirty [35]. It also emphasized the need for security verification of software dependencies at the

continuous monitoring level before they are deployed.

2.2.3 Codecov Attack (2021)

Codecov attack proved to be attacking CI/CD pipelines, undergoing a powerful and fundamental element of modern software development. Attackers hacked Codecov's Bash uploader script — a Bash tool that developers use for code coverage analysis. Affected projects had their sensitive environment variables such as API keys and credentials stolen from them and secretly sent to an attacker controlled server in the modified script [36]. The choice fell on this breach of the security risks of the third party integrations in the CI/CD pipeline. That was an alarm for organizations, to check very seriously to which extent was exposed through the use of external tools, how could they manage the credentials to minimize exposure [37].

2.2.4 Need for Stronger Supply Chain Security

As the activity of supply chain attack (SCA) has become more and more frequent as well as more sophisticated, protecting third party products has become a top security priority for teams. But, many organizations are not completely aware what dependencies they are using, meaning we are still vulnerable to future threats [38]. Companies should adopt the following security measures to disseminate these risks.

- Software Bill of Materials (SBOM): The ability to have a detailed inventory of all of the pieces of software within an application is used to be used to track the vulnerabilities within the application, but more importantly it helps with faster response when incidents occur in security.
- Automated Dependency Scanning: Snyk, Dependabot, and OWASP Dependency-Check continuously scan for known vulnerabilities in third party packages, giving people the ability to patch them before harm is done.
- Zero Trust Approach: The prudent attitude of an organization with respect to third party components is 'never trust, only verify'. By doing this, malicious code can be prevented to enter the software supply chain.
- Regular Code Audits and Penetration Testing: Frequent security assessments help detect anomalies at the earliest and in this way help preventing attacks done by them. Static analysis, code reviews and penetration tests should be expected in every software development cycle.
- Secure CI/CD Practices: Code signing in development pipelines, checking for artefact integrity and stringent access controls prevent unauthenticated, untrusted software components from being deployed.

2.3 Common Attack Vectors in Supply Chain Attacks

There are several forms of supply chain attacks and the techniques they use to compromise an apps' ecosystem. Unwanted and exploitable vulnerabilities in third party dependencies, development pipelines or even software distribution mechanisms are used by the cyber criminals to gain access to the systems without clearance [39]. Then are some of the most common attack vectors that are used in supply chain attacks.

2.3.1 Compromised Third-Party Libraries

Injection of malicious code in third party libraries is one of the most common ways of supply chain

attacks. Unfortunately for many users, attackers often gain this ability by hijacking a popular open source package or handing out a fake copy of a much loved library [40]. Cybercriminals in open-source package hijacking get access to maintainers' accounts with which they insert malicious code into update. However, when one of the developers unknowingly updates his dependencies including the compromised package, the package is automatically included in the applications that he himself developed. However, malicious packages are uploaded under the name of familiar packages in the JavaScript npm ecosystem to fool developers. Python's PyPI repository has seen dependency poisoning, where attackers glue altered versions of 'good' packages into hopes that untaught developers compulsively install them. The event-stream attack in npm is notorious example of this attack vector. Thousands of developers received the malicious code thanks to a new maintainer of the library that was inserted, with the aim to steal cryptocurrency wallet credentials [41].

2.3.2 Code Injection via Dependencies (Typo squatting and Dependency Confusion)

Secondly, attackers use package naming convention to distribute malicious dependencies by typo squatting and dependency confusion. The term typo squatting is used to describe when malicious packages are created with similar names to really popular libraries and hope that because it is common for developers to make typographical errors when installing dependencies, it will be used instead of the real package. For example, if a developer installs requests instead of requests, by mistake, the code being installed will contain the harmful code. Dependency confusion attacks exploit package managers' way of resolving dependencies [41]. An attacker can upload a public package with the same name as an organization's internal library (e.g. company-internal-library) stored in a private repo if it's an organization with an internal library (e.g. organization — has internal library in private repo) in a registry of npm or PyPI. Because package managers tend to put priority on public versions, developer may accidentally load the packages with the controller. Security researcher Alex Birsman found an ingenious technique that he successfully used to inject malicious dependencies into major tech companies like Apple, Microsoft and Tesla.

2.3.3 Backdoors in Open-Source Components

Open-source projects get some of the attackers to contribute seemingly legitimate code to open-source projects, which later you can exploit [35]. These backdoors can have months or years pass until they are activated. One notorious particular case involved UAParser.js attack where the hackers hacked a very widely used JavaScript library which does browser fingerprinting. The malware that got included into the package was meant to steal passwords and install cryptocurrency miners on infected PCs. The attack was detected and no longer possible after it was done because many applications and enterprises relied on UAParser.js.

2.3.4 Build System and CI/CD Pipeline Attacks

Build system and CI and CD pipeline attacks try to inject malicious code into the build or deployment process of the software development lifecycle [42]. Nowadays, attackers use vulnerabilities in CI/CD pipelines to control the software deployment, therefore it may be possible to inject malware into pro- duction environments [43]. It is possible for them to inject malicious scripts into, either build scripts or containerized software releases. Unauthorized access to someone's developer account or their code repository allows the attackers to commit a backdoor

International Journal of Core Engineering & Management
Volume-7, Issue-06, 2023 ISSN No: 2348-9510

into the source code, which they will be able to ex- ploit later. One such attack that serves as a prime example of all of this is the SolarWinds incident where attackers gained access into a trusted software update mechanism and subsequently caused widespread infiltration of government agencies and other major corporations.

2.4 Impact of Supply Chain Attacks

While supply chain attacks seem a hypothetical reality, the damage is real, and it can happen to organizations across many other industries [44]. This can affect financial losses, data breaches to complete service disruption. Businesses and governments are at risk of long-lasting damage from threats that stem from attackers breaking into software dependencies, development pipelines.

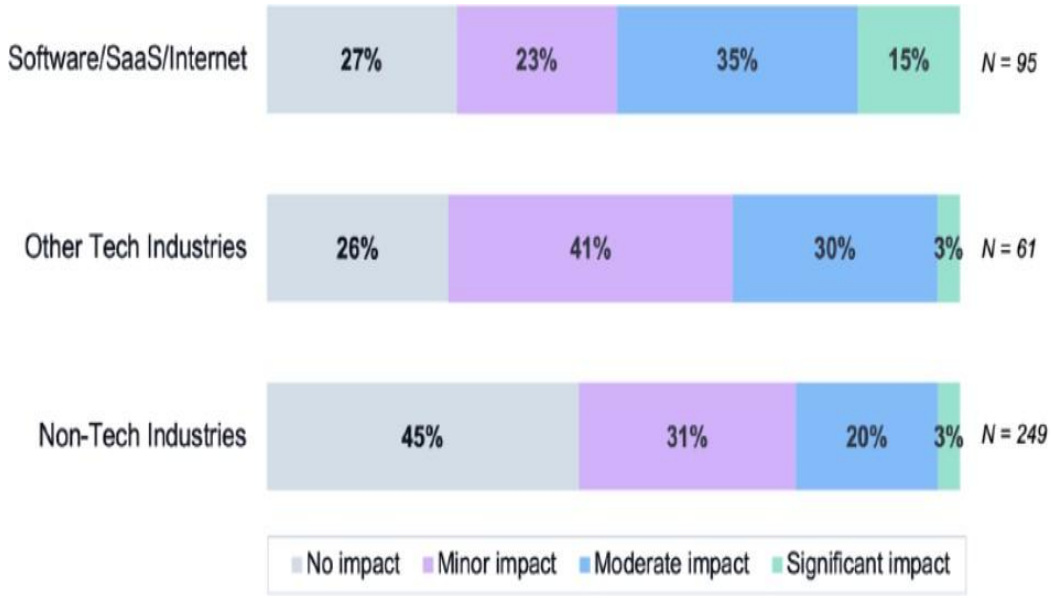

Figure 3: Impact of Supply Chain Attacks

2.4.1 Data Breaches and Information Theft

Attacks through compromised dependencies can lead to exfiltration of the sensitive data, including customer's personal details and payment records. Also, attackers may be able to enter inside the internal systems by authenticating credentials and API keys [45]. Financial and competitive disadvantages can also occur if intellectual property and proprietary code are stolen. CODECOW 2021 is a good example of exactly this: attackers inserted malicious code into CODECOW's Bash uploader script, and thus were able to obtain sensible environment variables from many companies in one go. This attack proves that developers tools and CI/CD pipes use must be safeguarded.

2.4.2 Remote Code Execution (RCE) and System Takeover

Attackers can infect their systems with malware, steal data or bring entire operations to their knees simply because the third-party libraries it runs are vulnerable. This threat is shown by Log4j (or Log4Shell) and Spring4Shell vulnerabilities, as simple HTTP requests allowed attackers to

remotely run commands on thousands of servers around the globe. The point here was that a single open-source vulnerability exposed could affect millions of systems, and this required proactive security measures in the software development process [46].

2.4.3 Financial and Reputational Damage

When affected, an organization often suffers significant financial losses in response to incident response, the audit of security, and system restoration costs [47]. If you expose customer data there are legal penalties that can result in hefty fines and can worsen your customer trust and adversely affect reputation, even rendering future receipt of revenue impossible. The Kaseya ransomware attack is a stark example of a cybercriminal taking advantage of Kaseya's remote management software's vulnerabilities. Once they were in–this allowed them to drop ransomware on to 3,000s of businesses around the world and ask for multimillions to unlock data.

2.4.4 Widespread Service Disruption

Disruption to critical applications and services due to compromised essential libraries or software components can result in operational failures, downtime, and, in the case of the OPENTM Librarian component, the potential for unpredictable behaviour and possibly failed transition to Dreamtime. It is particularly harmful to the industries like healthcare, where a compromised patient management system or hospital network can cause interruption in medical services, and finance, where banking applications and payment systems can be compromised, resulting in financial insecurity. The problem of cybercrime is just as much for government and defense sectors as it is for the industry thanks to the fact that cybercriminals can pose a serious risk to the national security due to targeting critical infrastructure [48]. A well understood example of such a disruption is the NotPetya attack, stemming from an attack on supply chain through the Ukrainian accounting software. Within a few years, it spread globally, causing billions of dollars' worth of damages and severely affecting major companies such as Maersk, FedEx, Merck.

III. CASE STUDIES: LOG4J AND SPRING4SHELL

Both the Log4j and Spring4Shell cases have highlighted the reality of how supply chain vulnerabilities can inflict devastation to a large number of organizations. These two vulnerabilities exposed very critical security holes in many used Java frameworks that allowed attackers to run remote code on affected machines [49]. These incidents were extremely serious, which put our focus to strengthen the security of third party dependencies and implement prompt mitigation.

3.1 Log4j (CVE-2021-44228)

The discovery of this Log4Shell (CVE-2021-44228) vulnerability kicked off a massive cybersecurity crisis centered around the widely used Java logging framework Log4j. This flaw existed in Log4j versions 2.0 through 2.14.1 due to the handling of user input in Java Naming and Directory Interface (JNDI) lookups supported by the framework. This feature can be exploited by attackers to execute arbitrary code on a target system and is one of the rarest remote code execution (RCE) vulnerabilities recently [50].

International Journal of Core Engineering & Management
Volume-7, Issue-06, 2023 ISSN No: 2348-9510

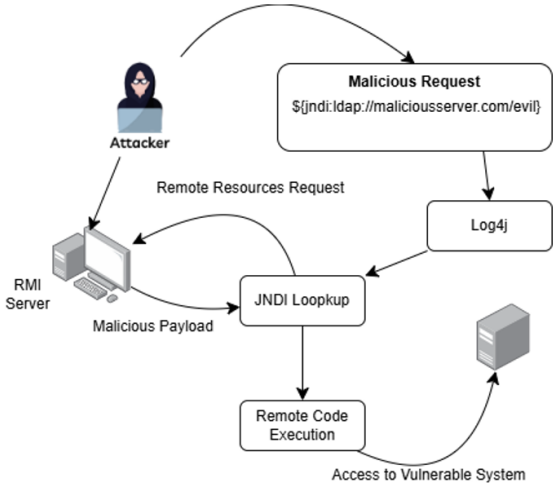

Figure 4: Log4j Exploitation

The Log4Shell exploit was rather simple to use. Any application which logs the user-controlled data that attackers can inject a specially crafted string into it. Log4j, when parsing a string containing a JNDI lookup command like $jndi:ldap://malicious-server.com/exploit would have made a request to the malicious server. This granted control of the affected system to the attacker, allowing it to be remotely executed and, in some cases, for the attacker to gain full control [51]. Log4Shell was a major concern because log4shell is a super easy thing to exploit in countless applications in all areas of life because loging is a basics feature with lots of applications.

Log4Shell impacted major cloud providers, enterprises and government agencies by taking down big companies. To prevent the impact of the vulnerability, cloud services like AWS, Microsoft Azure, and Google Cloud were required to react immediately. Nation state actors and cyber criminals were quick to turn the flaw into their own malicious tools such as data theft, cryptocurrency mining and ransomware. According to Microsoft and VMware, APT groups began using Log4Shell for espionage. Gangs of ransomware use the exploit to gain initial access to corporate networks and integrated it into attack chains.

Upon sensing the danger, organizations responded with speed and have put in place various security measures to avert the risk. Log4j 2.15.0, released by the Apache Software Foundation was disabled by default for JNDI lookups. Following that, additional patches were released to patch other weaknesses and make it more secure. Firewall rules, intrusion detection systems (IDS) and web application firewalls (WAFs) were deployed all over the world for blocking attempts at exploit. Many organizations created temporary workarounds by setting system properties to disable JNDI lookups while they worked at solving the issue on a more permanent level. Yet all of this has not prevented Log4Shell from remaining a threat because legacy applications and embedded devices remain unpatched. It was after the Log4Shell incident that the need of software supply chain security and proactive vulnerability management became even clearer. It was also a demonstration that even revered and relied upon software components of all sizes can have

destructive security vulnerabilities inside them, and they need to be monitored and updated constantly by organizations [52]. Tools to scan for and eliminate Log4shell have been developed by security researchers and vendors since and the long term implications on cybersecurity practices around this problem are still substantial.

3.2 Spring4Shell (CVE-2022-22965)

In March 2022, just a few months after the Log4Shell crisis, another significant Java vulnerability emerged: Spring4Shell (CVE-2022-22965). This vulnerability affected the Spring Framework, a widely used Java framework in enterprise applications and cloud environments [53]. Unlike Log4Shell, which stemmed from improper input validation in a logging library, Spring4Shell was triggered by insecure class binding mechanisms in Java, allowing attackers to achieve remote code execution when specific conditions were met. Specifically, Spring4Shell took advantage of a deserialization vulnerability in one of the data binding features in the Spring Core framework: it allows objects to be dynamically created with user input as parameters. On the remote machine, the attackers managed to have property values modified within the class loader scope to change certain of the application objects in order to run remote code. In a typical type of attack scenario, an HTTP request was sent that, however, changed the properties of the class loader. Attackers could upload a malicious web shell and obtain persistent remote access to compromised server by overwriting certain fields. However, most of the vulnerability existed in an application that is run on Apache Tomcat with JDK-9 or later. Sharing similarities with Log4Shell, Spring4Shell differed from the earlier vulnerability in some ways. Although Log4Shell was significantly easier to exploit, requiring nothing more than a simple string injection, in order for Spring4Shell to be vulnerable it Patches were released by the developers of the Spring Framework for versions 5.3.18 and 5.2.20, which restricted access of the class loader and prevented unauthorized modifications. More such security measures were taken by security teams which included disabling unsafe reflection, updating containerized applications to remove vulnerable components, and deploying web application firewalls for detection and blocking of the attacks. While Spring4Shell, contrary to Log4Shell, wasn't as bad, it reaffirmed the ongoing security issue with Java ecosystems and the necessity of continuous vulnerability management.

log4Shell and Spring4Shell remind us that the threat landscape is constantly evolving, with widely used software components that introduce very serious security risks. Such incidents have compelled regulatory bodies and cybersecurity frameworks to put ideal expeditious security measurements, utilizing software bill of materials (SBOM) following, and better application security testing, for the most part, to counteract a similar vulnerability from showing up in future. Log4Shell and Spring4Shell remind us that the threat landscape is constantly evolving, with widely used software components that introduce very serious security risks. Such incidents have compelled regulatory bodies and cybersecurity frameworks to put ideal expeditious security measurements, utilizing software bill of materials (SBOM) following, and better application security testing, for the most part, to counteract a similar vulnerability from showing up in future.

International Journal of Core Engineering & Management
Volume-7, Issue-06, 2023 ISSN No: 2348-9510

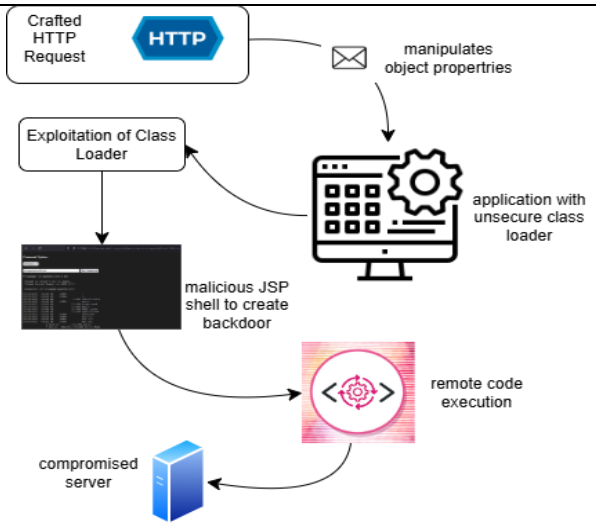

Figure 5: Spring4Shell Exploitation

IV. METHODOLOGIES TO IDENTIFY ISSUES

Proactive approach using automated tools, rigorous security assessment, and good practice in software development is needed for identifying and mitigating vulnerabilities in the software supply chain. As applications become more complex due to using open-source components as well as relying on third party dependencies, organizations must adopt comprehensive methodologies to maintain integrity of their applications. These methodologies are further concerned about the tracking of dependencies, scanning for vulnerabilities, and in detail code analysis to find and fix security holes before such holes can be exploited by the attackers. Organizations can thus best reduce their supply chain threat exposure and increase the security level of their software systems by employing these techniques.

4.1 Software Bill of Materials (SBOM)

A Software Bill of materials (SBOM) is a detailed inventory of all the software components and Third-party libraries used, use of dependency, etc, from an application. It gives visibility into the software supply chain, such that organizations can know the source, the versions, and the interrelationships of each component. As such, this transparency is key in assessing the security of software applications since it gives an organization the ability to quickly find out if they are using outdated or vulnerable components that attackers may exploit. Consequently, SBOMs are highly useful for reducing supply chain attack risks as they allow organizations to monitor dependencies and react quickly to newly detected vulnerabilities. By having an up-to-date SBOM in hand, security teams can logically compare the contents of their software components with vulnerability databases, such as the National Vulnerability Database (NVD) and act on any identified issue immediately. They also could automate generation and analysis of SBOMs into the software development lifecycle, which can continuously produce SBOMs for real time security assessments [54].

Since SBOMs are a necessary element of software security, the Cybersecurity and Infrastructure

Security Agency (CISA) has put forth guidelines to enable standards for the creation of SBOMs within specific industries. These guidelines propose machine readable SBOM formats, includ- ing SPDX (Software Package Data Exchange) and CycloneDX, to conduct an automated vulnerability detection and remediation process. In this focus, CISA has encouraged adoption of SBOM in federal software procurement, and it enriches the national cybersecurity resilience. Organisations can improve their ability to track and secure their software supply chain by implementing SBOMs and reduce the risks of supply chain attacks which could lead to supply chain attacks which would compromise critical systems and data

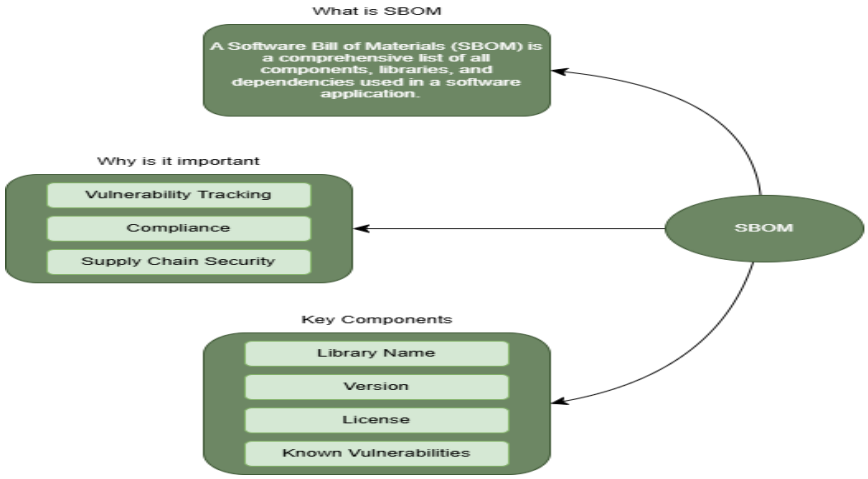

Figure 6: Software Bill of Materials (SBOM)

4.2 Dependency Scanning Tools

Dependency scanning tools are essential to identify vulnerabilities in those external dependencies that most modern applications rely on as they are dominated by third party libraries and open-source components. These tools are attuned to automatically analysing software dependencies and check them against known vulnerability databases to help an organization detect and fix security risks before they are exploited. Regular dependency scanning helps security teams to know any threats that can be unleashed and patches it accordingly. OWASP Dependency-Check is a widely used tool for Java based applications that scan dependencies being managed by JDK tools such as Maven and Gradle [55]. while also cross references dependencies with the National Vulnerability Database (NVD) to identify security flaws and supply intensive tell on probable threats. Likewise, tools such as Sonatype Lifecycle and Nexus IQ provide a comprehensive solution for dependency management for applications developed in JavaScript (NPM) and Python (PyPI). In addition to detecting vulnerabilities, these provide remediation guidance as well, and development teams can efficiently solve security problems without disrupting the development flow.

As containers move more into the mainstream, securing container dependencies has become a must. These types of tools include Snyk, GitHub Dependabot, and Trivy that used to detect vulnerabilities in containerized environments, scanning Docker images, Kubernetes deployments and infrastructure as code configuration. These tools are very easily integrated into CI/CD pipelines to make sure that we assess security at every stage of the software development lifecycle. Through dependency scanning tools, organizations can detect vulnerabilities proactively in its

software applications before the supply chain compromise can occur with grave consequence.

Tool Name	Supported Languages	Features	Integration with CI/CD
OWASP Dependency-Check	Java, Python, Maven JavaScript, etc	Identifies known vulnerabilities (CVEs)	Jenkins, GitHub Actions, GitLab, CLI
Snyk	Java, JavaScript, Python, Go, .NET, Rust, etc.	Real-time vulnerability detection, Fix suggestions, License compliance	GitHub, GitLab, Bitbucket, Jenkins, Azure
GitHub Depend-abot	JavaScript, Python, Ruby, Java, .NET, etc.	Automated dependency updates, CVE alerts	Built into GitHub (Pull Requests & Alerts)
GitLab Dependency Scanning	Java, JavaScript, Python, Ruby, etc.	Vulnerability scanning, Dependency insights	Integrated into Git-Lab CI/CD
Whitesource (Mend)	Multi-language support	Open-source risk management, Policy enforcement	Jenkins, GitHub, GitLab, Azure, CircleCI
Sonatype Nexus IQ	Maven, Java, JavaScript, Python, etc.	Advanced policy enforcement, Secu-rity analytics	Jenkins, GitHub, GitLab, Bitbucket

4.3 Static and Dynamic Code Analysis (SAST, DAST)

Static and dynamic code analysis are two important methodologies that facilitate identification of the security vulnerabilities in the organizations' software applications, before they are deployed in the pro- duction. Both of these approaches work in conjunction with one another rolling out a comprehensive security assessment that covers coding flaws as well as runtime vulnerabilities. Incorporating Static Application Security Testing (SAST) and Dynamic Application Security Testing (DAST) into the development pipeline allows companies to prevent and avoid the security weaknesses that arise throughout the software development lifecycle [56]. SAST is analysing the source code, bytecode or binaries without executing application. This way, security teams can find out if that belongs to hardcoded credentials, SQL injection threats, too much validation, and weak encryption even before the code is compiled or deployed. Real time feedback by SAST tools like Checkmarx, Fortify, SonarQube directly into development environments, aids in integration of SAST tools into development environments thus giving developers an opportunity to address security issues as they write code. SAST is one of the key advantages because it allows organizations to shift security left, allowing the detection of vulnerabilities early in the development cycle before they are harder and more expensive to fix.

While DAST aims at evaluating security vulnerabilities in a running application by staging real world attack scenarios, OSCAR performs the auditing in the source code without redeployment. While SAST examines the code structure, DAST tests how an application behaves under different scenarios by looking for vulnerabilities such as, for example, authentication flaws, session management weakness, cross site scripting (XSS), and API misconfigurations. For web applications

and APIs, popular DAST tools like Burp Suite and OWASP ZAP are commonly used to test and find exploitable weaknesses in the application that would not otherwise be apparent from static code analysis alone [57]. However, SAST and DAST are most effective when used together. DAST provides some insight into how an application behaves for real world scenarios while SAST is for detecting vulnerabilities at the code level. The amalgamation of both methodologies would help organizations in building a robust security framework to make their applications resistant against the ever-evolving cyber threats.

V. DEFENSIVE TECHNIQUES AND MITIGATION

5.1 Using Proxy Repositories

A proxy repository is one of the most effective ways to protect software supply chains by implementing an intermediary between external package repositories and developers. Proxy repositories add controlled access to dependencies through which only vetted and approved packages are downloaded and used for the distribution of the software within an organization's software development environment. Tools such as Sonatype Nexus and JFrog Artifactory allow enterprises to create local caches of third party libraries which reduces the chance their company is affected by a supply chain attack and also speeds up build performance by reducing the amount of external repository dependency [58].

Maven Central is a main source of dependencies for Java based applications and respectively using a proxy repository prevents untrusted libraries entering the development pipeline. Likewise, the NPM and PyPI proxying system is also crucial for the environments of JavaScript and Python prevent to facilitate direct download of malicious or compromised packages. Most viruses are injected into widely used libraries with malicious code and they aim to make it look like a legitimate update sold by vendors of these libraries, i.e., open source repositories. Organizations can make it entirely impossible for developers to accidentally bring in unverified dependencies by forcing them to use internal package repositories. Furthermore, proxy repositories provide auditing capabilities; security teams can see which versions of dependencies are being used and take steps such as enforcing policies on the use of libraries.

5.2 Securely Downloading Packages

Safe downloading practices lead to the security and integrity of software's dependencies in the first place. Package signatures are one such essential measure where cryptographic signatures are verified to ensure that the downloaded libraries are authentic and intact. Most modern package managers like Maven, NPM, and PyPI give signature verification built in. When this verification process fails for a package, developers should configure their environments so any such packages will be rejected, and thus tampered or malicious code will not end up in the software stack [59].

Organizations should go beyond signature verification and implement carefully designed allow lists which define what dependencies can be used. By doing this, only thoroughly reviewed and approved libraries can reach the application. Allow lists provide proactive control over what can be installed, unlike try to reactively deny known malicious components. Dependency management tools can be used by security teams to write policies, and enforce them, blocking any package that

has not explicitly been approved by you. Organizations can use signature verification plus allow lists to create a layered defense against attackers who deploy supply chain attacks stemming from untrusted or even compromised dependencies.

5.3 Implementing SBOM Best Practices

A Software Bill of Materials (SBOM) is a complete, machine readable list of dependencies for an application (libraries, frameworks and other third-party components). Having SBOM is a must; it allows you to track dependencies, identify vulnerabilities, as well as complete regulatory compliance. Thus, it offers full transparency into components in a software system to allow the program to determine and mitigate the potential security risks [60]. In order to produce an SBOM, organizations have options to utilize standardized formats like CycloneDX or SPDX that can be integrated natively in security monitoring technologies. SBOMs should be updated with the dynamically occurring inclusion of new dependencies or changes in dependencies. By having automated SBOM checks in CI/CD pipelines combined with real time alerts to report vulnerable or out of date components, developers can quickly, which hopefully will happen prior to deployment, be notified of a vulnerable or out of date component and remediate.

In line with CISA's call for adoption of SBOMs to make software supply chain security better, the agency has strongly pushed for SBOM use. Today, many regulatory frameworks and industry standards, including NIST's Secure Software Development Framework (SSDF), require the organization to have this up-to-date SBOM as part of their cybersecurity best practices. Through the use of SBOMs, organizations gain much more visibility into their application 'ecosystems' to better be able to proactively respond to emerging threats.

5.4 Regular Patch Management and Dependency Upgrades

Up to date dependencies is one of the most basic but neglected aspects of software security. Also, attackers will often use known vulnerabilities on outdated libraries to break into a system. But to avoid this risk, automated patch management solutions must be adopted by organizations that can guarantee that after a security update is published their software patch and deploy automatically [61]. Bot, one of the most widely used tool to manage dependency update, scans the projects continuously to find out outdated packages and automates the process of upgrading version. By using Renovate Bot or another tool in the CI/CD pipelines, software dependencies are kept up to date without manual intervention. Automatic updates should be carefully tested to avoid compatibility issues that may occur due to breaking changes.

Apart from automated updates, organizations should routinely perform security audits of their dependencies via programs such as OWASP Dependency Check, Snyk, and GitHub Dependabot. They broadcast alerts about vulnerable libraries to security teams who can use the severity and exploitability of a found library to know what to patch first. In addition, enterprises should create patch management policies, including which updates to do regularly, testing processes, and who to escalate to for handling critical vulnerabilities.

It is shown that by being proactive with patch management, organizations will significantly reduce their exposure to security threats, keeping their applications vulnerable to new attack techniques

until they are patched.

VI. CONCLUSION AND FUTURE DIRECTION

Such security risks are all too apparent with high profile vulnerabilities like Log4Shell (CVE-2021-44228) and Spring4Shell (CVE-2022-22965), which see increasing default reliance on open-source software and third-party dependencies. These incidents are proof that insecure dependency management can have dire results: seemingly trivial faults in popular libraries can result in remote code execution (RCE), data breaches, and widespread system compromise. In today's evolving software supply chain, organizations must proactively take security at risk under control to protect themselves.

The major takeaway from recent supply chain attacks is that simply automatic dependence management is not enough. The tools such as Maven, NPM, PyPI and others package managers significantly simplify the work process but also create entry point points for the attackers. Treat actors are turning directly to software repositories as a new target, infecting either malicious packages within popular repositories or using library exploits to attack widely adopted libraries. While rapidly exploiting Log4Shell and Spring4Shell illustrates why software dependencies must be made secure before attackers weaponise new vulnerabilities, the ability of attackers to find common vulnerabilities and use them against well-known applications suggests that the free sample rate could soon drop.

In attempts to combat these threats, organizations need to practice multi layered security measures. To prevent unauthorized package downloads and to have oversight of third-party dependencies, using proxy repositories can be helpful; they are Sonatype Nexus or JFrog Art factory, for example. Package signature verification and strict allow list prevent introducing components that do not have a verified signature to the software environment. Software Bill of Materials (SBOM) adoption also has the role of full transparency of all dependencies in an application used within it, which is essential for supply chain security. Automated SBOM checks on the CI/CD pipelines enable Organizations detect and remediate vulnerabilities before the software is deployed.

Another aspect of securing dependency management is being able to patch timely and to regularly reudoate. Many attackers rely on out-of-date software components with known vulnerabilities as a method of attack. Thus, patch management is a critical component to successful cybersecurity. Renovate Bot, OWASP Dependency-Check, Snyk, and GitHub Dependabot automate the detection of outdated libraries and security flaws to keep the organizations updated with the new threat in the industry. It's however important to manage automated updates carefully to avoid breaking dependencies and need for robust testing and version control makes. The current defensive approaches are largely about patching known vulnerabilities, and restricting access to untrusted dependencies, but there is the need of future research on dependency risk scoring and predictive vulnerability detection using AI. The machine learning models could take advantage of software repositories, developer activity and historical data of vulnerability from software to predict the most likely to be used in the future at some point. With the help of AI powered risk assessment

adds in CI/CD pipelines, an organization could be proactive in finding out about high-risk dependencies before they become catastrophic security threats. Furthermore, automated behavioural analysis of third party libraries can help catch change of bad behaviour in real time and prevent supply chain attacks like dependency confusion or typo squatting. Additional security can also be offered by the AI driven anomaly detection of malicious packages that appear in software package registries as legitimate updates.

REFERENCES

1. A. M. S. Laurent, understanding open source and free software licensing: guide to navigating licens- ing issues in existing & new software." O'Reilly Media, Inc.", 2004.
2. Z. Li, L. Tan, X. Wang, S. Lu, Y. Zhou, and C. Zhai, "Have things changed now? an empirical study of bug characteristics in modern open-source software," in Proceedings of the 1st workshop on Architectural and system support for improving software dependability, pp. 25–33, 2006.
3. K. Fogel, Producing open source software: How to run a successful free software project." O'Reilly Media, Inc.", 2005.
4. T. Hellstr¨om, "Critical infrastructure and systemic vulnerability: Towards a planning framework," Safety science, vol. 45, no. 3, pp. 415–430, 2007.
5. T. Kilamo, I. Hammouda, T. Mikkonen, and T. Aaltonen, "From proprietary to open source—growing an open-source ecosystem," Journal of Systems and Software, vol. 85, no. 7, pp. 1467–1478, 2012.
6. D. I. Board, "Software is never done: Refactoring the acquisition code for competitive advantage," Report of the Defense Innovation Board. Retrieved from https://media. defense. gov/2019/Mar/26/2002105909/-1/-1/0/SWAP. REPORT MAIN. BODY, vol. 3, p. 19, 2019.
7. M. Neovius, Trustworthy context dependency in ubiquitous systems. PhD thesis, Turku Centre for Computer Science (TUCS), 2012.
8. Nappa, R. Johnson, L. Bilge, J. Caballero, and T. Dumitras, "The attack of the clones: A study of the impact of shared code on vulnerability patching," in 2015 IEEE symposium on security and privacy, pp. 692–708, IEEE, 2015.
9. W. D. Eggers and P. Macmillan, The solution revolution: How business, government, and social enterprises are teaming up to solve society's toughest problems. Harvard Business Press, 2013.
10. M. Zalewski, The tangled Web: A guide to securing modern web applications. No Starch Press, 2011.
11. N. Selby and H. Vescent, The cyber-attack survival manual: tools for surviving everything from identity theft to the digital apocalypse. Weldon Owen International, 2017.
12. R. Bejtlich, The practice of network security monitoring: understanding incident detection and response. No Starch Press, 2013.
13. D. Waters, Supply chain risk management: vulnerability and resilience in logistics. Kogan Page Publishers, 2011.
14. Wilson, Botnets, cybercrime, and cyberterrorism: Vulnerabilities and policy issues for congress, vol. 29. Congressional Research Service Washington, DC, 2008.
15. M. Goodman, Future crimes: Everything is connected, everyone is vulnerable and what we can

do about it. Anchor, 2015.
16. M. A. Khair, "Security-centric software development: Integrating secure coding practices into the software development lifecycle," Technology & Management Review, vol. 3, no. 1, pp. 12–26, 2018.
17. L. Clark, Enterprise security: The manager's defense guide. Addison-Wesley Professional, 2003.
18. Payne, "On the security of open source software," Information systems journal, vol. 12, no. 1, pp. 61–78, 2002.
19. B. T. C. Palos, "Melhoria das pr´aticas de constru¸c˜ao de software: um caso de estudo," Master's thesis, Universidade de Aveiro (Portugal), 2012.
20. J. Newsome, D. Brumley, D. Song, J. Chamcham, and X. Kovah, "Vulnerability-specific execution filtering for exploit prevention on commodity software.," in NDSS, 2006.
21. G. Deepa and P. S. Thilagam, "Securing web applications from injection and logic vulnerabilities: Approaches and challenges," Information and Software Technology, vol. 74, pp. 160–180, 2016.
22. P. Eder-Neuhauser, T. Zseby, J. Fabini, and G. Vormayr, "Cyber attack models for smart grid environments," Sustainable Energy, Grids and Networks, vol. 12, pp. 10–29, 2017.
23. C. DiBona, M. Stone, and D. Cooper, Open sources 2.0: The continuing evolution. " O'Reilly Media, Inc.", 2005.
24. L. Cazorla, C. Alcaraz, and J. Lopez, "Cyber stealth attacks in critical information infrastructures," IEEE Systems Journal, vol. 12, no. 2, pp. 1778–1792, 2016.
25. P. A. Gloor, Swarm creativity: Competitive advantage through collaborative innovation networks. Oxford University Press, 2006.
26. M. Rashid, P. M. Clarke, and R. V. O'Connor, "A systematic examination of knowledge loss in open-source software projects," International Journal of Information Management, vol. 46, pp. 104–123, 2019.
27. R. J. Anderson, Security engineering: a guide to building dependable distributed systems. John Wiley & Sons, 2010.
28. Brown and G. Wilson, The Architecture of Open Source Applications: Elegance, Evolution, and a Few Fearless Hacks, vol. 1. Lulu. com, 2011.
29. O. Pieczul, S. Foley, and M. E. Zurko, "Developer-centered security and the symmetry of ignorance," in Proceedings of the 2017 New Security Paradigms Workshop, pp. 46–56, 2017.
30. G. Hu, X. Yuan, Y. Tang, and J. Yang, "Efficiently, effectively detecting mobile app bugs with appdoctor," in Proceedings of the Ninth European Conference on Computer Systems, pp. 1–15, 2014.
31. M. Jakobsson and Z. Ramzan, Crimeware: understanding new attacks and defenses. Addison-Wesley Professional, 2008.
32. Skoudis and L. Zeltser, Malware: Fighting malicious code. Prentice Hall Professional, 2004.
33. Singh and K. Chatterjee, "Cloud security issues and challenges: A survey," Journal of Network and Computer Applications, vol. 79, pp. 88–115, 2017.
34. S. Paquette, P. T. Jaeger, and S. C. Wilson, "Identifying the security risks associated with governmental use of cloud computing," Government information quarterly, vol. 27, no. 3, pp. 245–253, 2010.
35. Hoglund and G. McGraw, Exploiting software: How to break code. Pearson Education India,

2004.
36. B. Lakshmiraghavan, "Security vulnerabilities," in Pro ASP. NET Web API Security: Securing ASP. NET Web API, pp. 345–373, Springer, 2013.
37. D. Cappelli, A. Moore, R. Trzeciak, and T. J. Shimeall, "Common sense guide to prevention and detection of insider threats," 2009.
38. C. P. Pfleeger and S. L. Pfleeger, Analyzing computer security: A threat/vulnerability/countermeasure approach. Prentice Hall Professional, 2012.
39. K. Huang, M. Siegel, and S. Madnick, "Systematically understanding the cyber attack business: A survey," ACM Computing Surveys (CSUR), vol. 51, no. 4, pp. 1–36, 2018.
40. K. D. Mitnick and W. L. Simon, The art of intrusion: the real stories behind the exploits of hackers, intruders and deceivers. John Wiley & Sons, 2009.
41. R. K. Vaidya, L. De Carli, D. Davidson, and V. Rastogi, "Security issues in language-based software ecosystems," arXiv preprint arXiv:1903.02613, 2019.
42. C. Paule, "Securing devops: detection of vulnerabilities in cd pipelines," Master's thesis, 2018.
43. M. Koopman, "A framework for detecting and preventing security vulnerabilities in continuous integration/continuous delivery pipelines," Master's thesis, University of Twente, 2019.
44. M. Knemeyer, W. Zinn, and C. Eroglu, "Proactive planning for catastrophic events in supply chains," Journal of operations management, vol. 27, no. 2, pp. 141–153, 2009.
45. P. Siriwardena, "Advanced api security," Apress: New York, NY, USA, 2014.
46. J. A. Ozment, Vulnerability discovery & software security. PhD thesis, University of Cambridge, 2007.
47. P. Purpura, Security and loss prevention: An introduction. Butterworth-Heinemann, 2007.
48. E. Haber and T. Zarsky, "Cybersecurity for infrastructure: a critical analysis," Fla. St. UL Rev., vol. 44, p. 515, 2016.
49. D. Malkhi and M. K. Reiter, "Secure execution of java applets using a remote playground," IEEE Transactions on Software Engineering, vol. 26, no. 12, pp. 1197–1209, 2000.
50. B. Seri, G. Vishnepolsky, and D. Zusman, "Critical vulnerabilities to remotely compromise vxworks, the most popular rtos," White Paper, ARMIS, URGENT/11, 2019.
51. S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer, "Non-control-data attacks are realistic threats.," in USENIX security symposium, vol. 5, p. 146, 2005.
52. D. A. Fernandes, L. F. Soares, J. V. Gomes, M. M. Freire, and P. R. In´acio, "Security issues in cloud environments: a survey," International journal of information security, vol. 13, pp. 113–170, 2014.
53. C. Cimpanu, "Spring4Shell: New zero-day vulnerability uncovered in Spring Framework," The Record by Recorded Future, March 31, 2022.
54. X. Ding, F. Zhao, L. Yan, and X. Shao, "The method of building sbom based on enterprise big data," in 2019 3rd International Conference on Electronic Information Technology and Computer Engineering (EITCE), pp. 1224–1228, IEEE, 2019.
55. Y. Wang, B. Chen, K. Huang, B. Shi, C. Xu, X. Peng, Y. Wu, and Y. Liu, "An empirical study of usages, updates and risks of third-party libraries in java projects," in 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 35–45, IEEE, 2020.
56. Y. Pan, "Interactive application security testing," in 2019 International Conference on Smart Grid and Electrical Automation (ICSGEA), pp. 558–561, IEEE, 2019.

57. T. Rangnau, R. v. Buijtenen, F. Fransen, and F. Turkmen, "Continuous security testing: A case study on integrating dynamic security testing tools in ci/cd pipelines," in 2020 IEEE 24th Interna- tional Enterprise Distributed Object Computing Conference (EDOC), pp. 145–154, IEEE, 2020.
58. Y. Ma, "Software supply chain development and application," 2020.
59. W. Ozga, D. L. Quoc, and C. Fetzer, "A practical approach for updating an integrity-enforced operating system," in Proceedings of the 21st International Middleware Conference, pp. 311–325, 2020.
60. M. Buchheit, M. Hermeling, F. Hirsch, B. Martin, and S. Rix, "Software trustworthiness best practices," Industrial Internet Consortium, 2020.
61. R. Duan, A. Bijlani, Y. Ji, O. Alrawi, Y. Xiong, M. Ike, B. Saltaformaggio, and W. Lee, "Automating patching of vulnerable open-source software versions in application binaries.," in NDSS, 2019.