Build and Runtime Integrity for Java

Aman Sharma
KTH Royal Institute of Technology
Stockholm, Sweden
amansha@kth.se

ABSTRACT

Software Supply Chain attacks are increasingly threatening the security of software systems, with the potential to compromise both build and runtime integrity. Build-time integrity ensures that the software artifact creation process, from source code to compiled binaries, remains untampered. Runtime integrity, on the other hand, guarantees that the executing application loads and runs only trusted code, preventing dynamic injection of malicious components. This paper explores solutions to safeguard Java application's software supply chain at both stages. We propose techniques to detect malicious code injection through two main contributions: (1) novel algorithm for Java artifact equivalence, and (2) detection and prevention of runtime code injection.

KEYWORDS

Software Supply Chain, Java, Build Integrity, Runtime Integrity

1 INTRODUCTION

The software supply chain refers to the sequence of steps and inputs resulting in the creation of a software artifact [16]. The steps encompass everything from source code development to the compilation, packaging, and distribution of the final artifact which can either be executed or be reused for further development. While the supply chain facilitates efficient software production and deployment, it also introduces points of vulnerability that attackers can exploit, leading to what is known as a *software supply chain attack*. Such attacks aim to infiltrate the supply chain at any stage, by injecting malicious code and eventually compromising the software artifact. What makes these attacks dangerous is their cascading impact: by compromising a single step in the supply chain, attackers can affect not only the direct users of that component but also all downstream projects that depend on it.

Recent years have seen an alarming rise in software supply chain attacks [23]. According to Sonatype's 2024 report, the number of attacks detected are growing exponentially [17]. One of the most notable incidents is the Log4Shell vulnerability in the widely used Log4j logging library [22]. This vulnerability allowed attackers to execute arbitrary code remotely when the server logs a malicious input. Runtime security vulnerabilities like Log4Shell are particularly dangerous because they exploit the dynamic nature of modern applications. In Java-based systems, dynamic class loading is present [13] to load classes at runtime but it can also create potential attack surfaces. For instance, when a Java application loads classes or resources at runtime, attackers can potentially redirect classloading lookups to malicious sources, injecting harmful code into a running application and this is the case for Log4Shell. Traditional static analysis tools would not be able to detect such malicious behavior because it is only revealed at runtime [11]. Since this attack is in the Java ecosystem, it incurred significant damage as Java is widely used in critical infrastructure, financial systems, and large-scale enterprise applications [3]. Another example is the SolarWinds attack, where the build toolchain of SolarWinds was compromised, leading to the distribution of a malicious update to thousands of organizations [24]. These incidents underscore the critical need to secure the software supply chain during build time and runtime.

In this paper, we present 2 planned solutions for securing the software supply chain of Java applications: 'Algorithm for Java artifact equivalence' and 'Detection and prevention of runtime code injection'. The former contributes to build integrity while the latter contributes to runtime integrity.

2 PROBLEM STATEMENT

The main objective of my PhD project is to improve the state of the art for securing software supply chain in Java. We specifically address the problem of injection of malicious code in software artifacts at both build time and runtime. We choose Java because of its widespread adoption in enterprise and government systems [3]. 30% of developers vote Java as the most popular language [18]. Moreover, Java's extensive dependency ecosystem, with Maven Central hosting millions of artifacts, means that a single compromised component can potentially affect thousands of downstream applications resulting in devastating software supply chain attacks. The combination of Java's widespread use in critical systems, and its complex dependency network makes it both an attractive target for attackers and a crucial focus for supply chain security research. Thus, we address threats to build integrity and runtime integrity in Java.

2.1 Build Integrity for Java

Build Integrity means that the software artifact on package registries corresponds to its source code [25]. It is an important property as it ensures that the software artifact has not been tampered with during the build process by the compiler, continuous integration or delivery systems, or any other tool that interacts with source code. This property assures the third-party users that their binaries are built from the same source the supplier claim [1]. SolarWinds attack is an infamous example of a build integrity violation where the build toolchain of SolarWinds was compromised [24] which led to malicious updates to thousands of downstream users. Gruhn et al. [7] show that public continuous integration systems can inject malicious code in the artifact and the artifact can be downloaded by unsuspecting users. Ken Thompson also demonstrates that a compiler can inject malicious code in the artifacts it produces [20] even though its source code would not be malicious.

The key challenge for build integrity is the fact that current solutions do bit by bit comparison of artifacts [10] to ensure that the build has not been tampered with. This comparison approach is formally called Reproducible Builds [1] which is a technique to verify that the same source code with the same build environment produces semantically identical artifact. However, the bit-by-bit comparison is too strict as the the compilation process introduces spurious changes that are not relevant to the semantics of the program [8] [6]. For example, the order of constant pool entries in Java bytecode can differ between two builds of the same source code [26]. This causes the builds to be classified as different even though they are semantically equivalent. Schott et al. [14] propose a normalization tool that removes such differences. However, it focuses on only on Java bytecode across different Java versions and does not take care of normalizing other features such as timestamps, file permissions, etc. Thus, in this thesis proposal, we want to address the problem of detecting semantically equivalent and non-equivalent code between two builds and hence, relaxing the definition of Reproducible Builds.

2.2 Runtime Integrity for Java

Runtime Integrity means that the software is executing as expected based on its stack trace, heap, threads, and loaded code [19]. Any modification of execution behavior in terms of above properties can be a runtime integrity violation. An infamous example of such violation is the Log4Shell attack [22] where an external and malicious code is loaded into the system. In Java, dynamic classloading via JNDI lookup, Nashorn JavaScript engine, `Class.forName`, and `URLClassloader` can trigger such malicious code injection.

The key challenge for checking runtime integrity is that all approaches entail modifying the application, or its runtime environment or they require manual work. In related work, there are three approaches for checking runtime integrity. First, permission-based access controls for each dependency [2]. However, this approach requires modifications in the runtime and they differ for each version of the runtime. Hence, it is not scalable. Second, compartmentalization of dependencies in a separate runtime environment [9]. This approach suffers from overhead because of context switching between compartments and requires manual work to split code. Finally, the third approach relies on measuring integrity in terms of execution behavior [19]. This approach creates policy in the form of predicates over execution behavior and detects malicious code at runtime. However, each policy is tailored to an application and thus also requires manual work. In this thesis, we want to contribute to the runtime integrity of Java with new techniques for detecting malicious code injected at runtime.

3 THESIS CONTRIBUTIONS

The expected contributions of this thesis are two solutions to secure the software supply chain of Java at build time and runtime.

(1) A novel approach to analyze Java artifacts for verifying reproducibility. (Build Integrity)
(2) A novel approach to detect and prevent malicious code injected at runtime (Runtime Integrity).

3.1 Novel algorithm for Java artifact equivalence

We propose our first contribution which is a novel approach to analyze Java `artifact` for verifying reproducibility. The main idea is to create an intermediate representation that would only encapsulate the semantic features of an artifact. Thus, ignoring the spurious differences inserted by Java compiler. This algorithm can be applied to pair of Java artifacts to check if they are semantically equivalent.

To evaluate this novel algorithm, we plan to run it on a dataset of known reproducible Java artifacts and show that we can detect semantically equivalent code more accurately. The dataset is provided by `Reproducible Central` [4] which is an infrastructure on GitHub that builds and verifies reproducible Java artifacts. To prove the effectiveness of our algorithm, we compare the results with the baseline tool `jNorm` [14]. `jNorm` [14] is a tool that canonicalizes Java bytecode by ignoring differences that are because of differences in Java compiler versions or vendors. Hence, the baseline tool is apt for comparison as it is designed to ignore irrelevant differences in Java bytecode. Finally, we show the soundness of our algorithm by ensuring that it can also identify semantically non-equivalent artifacts. BinEQ [5] is a benchmark for binary equivalence that has pairs of semantically non-equivalent bytecode and we show that our algorithm can detect the non-equivalence.

3.2 Detection and prevention of runtime code injection

Our second contribution is a novel approach to detect malicious code injected at runtime. Its core novelty is that it does not require any modifications to the application, its runtime environment, nor any manual work. It is a two step fully automatic approach to detect malicious code injected at runtime and prevent it from execution. First, it builds an allowlist of classes that are allowed to be executed at runtime. Second, it intercepts all classes being loaded at runtime and checks whether they exist in the allowlist. `SBOM.exe` [15] is a prototype of the tool that we plan to evaluate.

We evaluate `SBOM.exe` by showing that it is effective in mitigating software supply chain attacks, compatible with real-world Java applications, and incurs low runtime overhead. The effectiveness is evaluated by showing that it can detect and prevent the execution of malicious code injected at runtime. We replicate the Log4Shell attack and showed that `SBOM.exe` can detect and prevent its execution. Next, we curate a dataset of real-world Java applications and show that `SBOM.exe` can be integrated into them without any modifications to the application or the Java runtime. Finally, we show that `SBOM.exe` incurs low runtime overhead for integrity checking. We create benchmarks with Java Microbenchmark Harness (JMH) [12] to measure only the runtime of application without the application warmup time [21].

4 CONCLUSION

We address the problem of injection of malicious code in software artifacts at both build time and runtime. Our first contribution is a novel approach to analyze Java artifacts for verifying reproducibility. This algorithm will help ensure that the software artifact has not been tampered with during the build process. Our second contribution is a novel approach to detect malicious code injected at runtime. This technique will help ensure that the software is executing only the trusted code.

REFERENCES

[1] 2024. Reproducible Builds — a Set of Software Development Practices That Create an Independently-Verifiable Path from Source to Binary Code. https://reproducible-builds.org/

[2] Paschal C. Amusuo, Kyle A. Robinson, Tanmay Singla, Huiyun Peng, Aravind Machiry, Santiago Torres-Arias, Laurent Simon, and James C. Davis. 2024. ZTD$_{JAVA}$: Mitigating Software Supply Chain Vulnerabilities via Zero-Trust Dependencies. https://doi.org/10.48550/arXiv.2310.14117 arXiv:2310.14117

[3] Alexander Belokrylov. 2022. Council Post: Why And How Java Continues To Be One Of The Most Popular Enterprise Coding Languages. https://www.forbes.com/councils/forbestechcouncil/2022/04/06/why-and-how-java-continues-to-be-one-of-the-most-popular-enterprise-coding-languages/

[4] Hervé Boutemy. 2024. Jvm-Repo-Rebuild/Reproducible-Central. jvm-repo-rebuild. https://github.com/jvm-repo-rebuild/reproducible-central

[5] Jens Dietrich, Tim White, Mohammad Abdollahpour, Elliott Wen, and Behnaz Hassanshahi. 2024. BinEq-A Benchmark of Compiled Java Programs to Assess Alternative Builds. https://www.researchgate.net/publication/383666359_BinEq-A_Benchmark_of_Compiled_Java_Programs_to_Assess_Alternative_Builds

[6] Jens Dietrich, Tim White, Behnaz Hassanshahi, and Paddy Krishnan. 2024. Levels of Binary Equivalence for the Comparison of Binaries from Alternative Builds. https://doi.org/10.48550/arXiv.2410.08427 arXiv:2410.08427

[7] Volker Gruhn, Christoph Hannebauer, and Christian John. 2013. Security of Public Continuous Integration Services. In *Proceedings of the 9th International Symposium on Open Collaboration (WikiSym '13)*. Association for Computing Machinery, New York, NY, USA, 1–10. https://doi.org/10.1145/2491055.2491070

[8] Irfan Ul Haq and Juan Caballero. 2021. A Survey of Binary Code Similarity. *ACM Comput. Surv.* 54, 3 (April 2021), 51:1–51:38. https://doi.org/10.1145/3446371

[9] Jianyu Jiang, Xusheng Chen, TszOn Li, Cheng Wang, Tianxiang Shen, Shixiong Zhao, Heming Cui, Cho-Li Wang, and Fengwei Zhang. 2020. Uranus: Simple, Efficient SGX Programming and Its Applications. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security (ASIA CCS '20)*. Association for Computing Machinery, New York, NY, USA, 826–840. https://doi.org/10.1145/3320269.3384763

[10] Chris Lamb and Stefano Zacchiroli. 2022. Reproducible Builds: Increasing the Integrity of Software Supply Chains. *IEEE Software* 39, 2 (March 2022), 62–70. https://doi.org/10.1109/MS.2021.3073045

[11] Marc Ohm, Henrik Plate, Arnold Sykosch, and Michael Meier. 2020. Backstabber's Knife Collection: A Review of Open Source Software Supply Chain Attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, Clémentine Maurice, Leyla Bilge, Gianluca Stringhini, and Nuno Neves (Eds.). Springer International Publishing, Cham, 23–43. https://doi.org/10.1007/978-3-030-52683-2_2

[12] Oracle. 2023. OpenJDK: Jmh. https://openjdk.org/projects/code-tools/jmh/

[13] Oracle. 2023. URLClassLoader (Java SE 21 & JDK 21). https://docs.oracle.com/en%2Fjava%2Fjavase%2F21%2Fdocs%2Fapi%2F%2F/java.base/java/net/URLClassLoader.html

[14] Stefan Schott, Serena Elisa Ponta, Wolfram Fischer, Jonas Klauke, and Eric Bodden. 2024. Java Bytecode Normalization for Code Similarity Analysis. In *DROPS-IDN/v2/Document/10.4230/LIPIcs.ECOOP.2024.37*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. https://doi.org/10.4230/LIPIcs.ECOOP.2024.37

[15] Aman Sharma, Martin Wittlinger, Benoit Baudry, and Martin Monperrus. 2024. SBOM.EXE: Countering Dynamic Code Injection Based on Software Bill of Materials in Java. https://doi.org/10.48550/arXiv.2407.00246 arXiv:2407.00246 [cs]

[16] SLSA. 2024. Terminology. https://slsa.dev/spec/v1.0/terminology

[17] Sonatype. 2024. 2024 State of the Software Supply Chain | Executive Summary. https://www.sonatype.com/state-of-the-software-supply-chain/introduction

[18] StackOverflow. 2024. Technology | 2024 Stack Overflow Developer Survey. https://survey.stackoverflow.co/2024/technology/

[19] Mark Thober, J. Aaron Pendergrass, and Andrew D. Jurik. 2012. JMF: Java Measurement Framework: Language-Supported Runtime Integrity Measurement. In *Proceedings of the Seventh ACM Workshop on Scalable Trusted Computing (STC '12)*. Association for Computing Machinery, New York, NY, USA, 21–32. https://doi.org/10.1145/2382536.2382542

[20] Ken Thompson. 1984. Reflections on Trusting Trust. *Commun. ACM* 27, 8 (Aug. 1984), 761–763. https://doi.org/10.1145/358198.358210

[21] Luca Traini, Vittorio Cortellessa, Daniele Di Pompeo, and Michele Tucci. 2022. Towards Effective Assessment of Steady State Performance in Java Software: Are We There Yet? *Empirical Software Engineering* 28, 1 (Nov. 2022), 13. https://doi.org/10.1007/s10664-022-10247-x

[22] Ilkka Turunen. 2021. Log4shell by the Numbers- Why Did CVE-2021-44228 Set the Internet on Fire? https://www.sonatype.com/blog/why-did-log4shell-set-the-internet-on-fire

[23] verizon. 2024. *2024 Data Breach Investigations Report*. Technical Report. https://www.verizon.com/business/resources/T169/reports/2024-dbir-data-breach-investigations-report.pdf

[24] Vijay A. D'Souza. 2021. SolarWinds Cyberattack Demands Significant Federal and Private-Sector Response (Infographic) | U.S. GAO. https://www.gao.gov/blog/solarwinds-cyberattack-demands-significant-federal-and-private-sector-response-infographic

[25] Duc-Ly Vu, Fabio Massacci, Ivan Pashchenko, Henrik Plate, and Antonino Sabetta. 2021. LastPyMile: Identifying the Discrepancy between Sources and Packages. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 780–792. https://doi.org/10.1145/3468264.3468592

[26] Jiawen Xiong, Yong Shi, Boyuan Chen, Filipe R. Cogo, and Zhen Ming (Jack) Jiang. 2022. Towards Build Verifiability for Java-based Systems. In *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP '22)*. Association for Computing Machinery, New York, NY, USA, 297–306. https://doi.org/10.1145/3510457.3513050