Indian Journal of Computer Science and Technology
https://www.doi.org/10.59256/indjcst.20260502106
Volume 5, Issue2 (May-August 2026), PP: 972-978.
www.indjcst.com
ISSN No: 2583-5300

A Systematic Review of Java Security: Architecture, Cryptographic Services, Vulnerabilities, and Emerging Security Paradigms

Md. Aali Rahman[1], Khushboo Rani[2], Md. Irfan Alam[3]
[1]*Bachelors of Computer Science and Engineering (AI & ML),Woxsen University, Hyderabad, Telangana, India.*
[2]*Assistant professor, Dr. Shyama Prasad Mukherjee University, Ranchi, Jharkhand, India.*
[3]*Faculty of CSE & IT, Jharkhand Rai University, Ranchi, Jharkhand, India.*

To Cite this Article: *Md. Aali Rahman[1], Khushboo Rani[2], Md. Irfan Alam[3], "A Systematic Review of Java Security: Architecture, Cryptographic Services, Vulnerabilities, and Emerging Security Paradigms", Indian Journal of Computer Science and Technology, Volume 05, Issue 02 (May-August 2026), PP: 972-978.*

Copyright: ©2026 This is an open access journal, and articles are distributed under the terms of the Creative Commons Attribution License; Which Permits unrestricted use, distribution, and reproduction in any medium, provided the original author and source are credited.

Abstract: *Java continues to play a significant role in the development of enterprise software, mobile applications, and embedded solutions because of its portability, reliability, and extensive security capabilities. This review investigates the security ecosystem of the Java platform by examining both its built-in protection mechanisms and the security challenges faced by modern Java applications. The study begins with an analysis of the fundamental components of Java's security model, including the Java Virtual Machine (JVM), bytecode validation processes, class-loading mechanisms, and the Security Manager, whose role has diminished in recent Java releases. In addition, the review discusses the Java Cryptography Architecture (JCA), emphasizing its support for widely used security technologies such as the Advanced Encryption Standard (AES), Rivest–Shamir–Adleman (RSA) encryption, SHA-256 hashing, secure random number generation, and Transport Layer Security (TLS) protocols for protected communication. The paper further investigates common vulnerabilities observed in Java-based systems. Particular attention is given to issues such as SQL injection, unsafe deserialization practices, misuse of reflection, exposure of hard-coded credentials, and reliance on outdated cryptographic algorithms. The causes of these weaknesses, their potential consequences, and recommended countermeasures are examined with reference to established security standards and current academic research. In addition, the review outlines key secure software development practices that can strengthen application security. These include parameterized database queries, rigorous input validation and sanitization techniques, adherence to the principle of least privilege, and systematic management of third-party libraries and dependencies throughout the software lifecycle. The paper also discusses recent developments shaping the future of Java security. Topics include the gradual retirement of the Security Manager, the increasing importance of the Java Platform Module System (JPMS) for application modularization and isolation, security considerations in cloud-native deployments, and the growing application of artificial intelligence for automated vulnerability identification and threat analysis. By consolidating findings from contemporary literature, these reviews offers insights into Java's evolving security landscape and presents practical guidance for building resilient and secure Java applications.*

Keywords: *Java Security, Java Virtual Machine, Byte code Verification, Cryptography, SQL injection, Secure Coding.*

I. INTRODUCTION

With the rapid growth of digital technologies and interconnected systems, security has become an essential aspect of software development. Organizations increasingly depend on software applications to manage critical operations, making them attractive targets for cyber threats. Security flaws within applications can lead to data breaches, financial losses, regulatory penalties, and damage to organizational reputation. As a result, modern software development requires strong security mechanisms not only at the application level but also within the underlying programming platforms and execution environments. Numerous security incidents have demonstrated that weaknesses such as code injection, memory-related vulnerabilities, and unsafe object handling can emerge when security controls are inadequately implemented or improperly configured [1, 2].

Among modern programming languages, Java has long been recognized for its emphasis on secure software execution. Since its introduction, Java has incorporated several protective mechanisms, including a managed runtime environment, automatic garbage collection, and multiple layers of security controls. These features help reduce common programming errors and provide a controlled environment for executing applications, including code obtained from external or potentially untrusted sources [1, 2].

Java's popularity extends across a wide spectrum of computing environments, ranging from enterprise information systems and web applications to mobile platforms and cloud-based services. Its platform independence, supported by the principle of "Write Once, Run Anywhere," has enabled developers to deploy applications across diverse operating systems with minimal modification. Furthermore, the availability of mature frameworks, development tools, and extensive community support has strengthened Java's position in sectors such as finance, e-commerce, healthcare, and large-scale distributed computing [3,4].

Despite these advantages, Java applications continue to face evolving security challenges. Attackers frequently exploit weaknesses associated with insecure deserialization, vulnerable third-party libraries, reflection misuse, and dependency-related issues. In recent years, researchers and practitioners have reported numerous incidents in which such vulnerabilities were leveraged to compromise application security, emphasizing the need for continuous assessment and improvement of Java-based systems [9, 10].

This review aims to provide a comprehensive understanding of Java security from both architectural and practical perspectives. It begins by examining the key elements of Java's security infrastructure, including the Java Virtual Machine (JVM), byte code verification process, class-loading mechanisms, and the Security Manager. The study then discusses the cryptographic facilities available through the Java Cryptography Architecture (JCA) and their contribution to protecting data confidentiality, integrity, authentication, and secure communication channels [5,6]. In addition, commonly reported vulnerabilities in Java applications are analyzed, together with mitigation approaches recommended by OWASP and supported by contemporary research findings [7,8]. The paper also considers recent developments that are shaping the future of Java security, including the gradual transition away from the Security Manager, the increasing adoption of the Java Platform Module System (JPMS), and emerging security requirements in cloud-native computing environments [11, 12].

II.JAVA SECURITY ARCHITECTURE

2.1 Java Virtual Machine (JVM)

The Java Virtual Machine (JVM) serves as the foundation of the Java platform, providing a runtime environment in which Java applications are executed. rather than running directly on the host operating system, Java programs are compiled into an intermediate representation known as bytecode, which is interpreted or compiled by the JVM. This architecture enables platform independence while also contributing to application security [1].

A key advantage of the JVM is its ability to perform multiple security-related checks before and during program execution. These checks include validating bytecode instructions, managing memory access, enforcing access restrictions, and monitoring runtime behavior. By acting as an intermediary between the application and the underlying system, the JVM helps reduce the risk of unauthorized operations and limits the impact of potentially harmful code [1,2].

The security philosophy behind Java follows a layered approach in which multiple mechanisms work together to protect the execution environment. Language-level safeguards, bytecode validation procedures, class-loading controls, and access-management policies collectively create several barriers against attacks. This multi-layered design ensures that the failure of a single security component does not necessarily compromise the overall security of the system [1, 3].

2.2 Bytecode Verification

One of the most important security features of the Java platform is the bytecode verification process. Before a class is executed, the JVM examines its bytecode to ensure that it adheres to the structural and type-safety requirements defined by the Java specification. The verifier checks whether instructions are used correctly, whether data types are consistent, and whether stack operations are performed in a valid manner [2].

This verification stage helps prevent a wide range of errors and security threats that could otherwise arise from malformed or intentionally manipulated bytecode. It ensures that object references are handled properly, variables are accessed according to their declared types, and program execution follows established safety rules. As a result, Java applications can execute code obtained from external or third-party sources with a higher level of confidence and reliability [2].

Over time, the verification mechanism has evolved from a conceptual design into a rigorously studied process supported by formal analysis techniques. Researchers have developed mathematical models and verification methods to demonstrate the correctness and effectiveness of byte code verification, reinforcing its role as a critical component of Java's security framework [2].

2.3Class Loader Mechanism: Java’s class-loading architecture is hierarchical: the bootstrap loader loads core platform classes, the platform loader loads standard extensions, and application class loaders load user code, with each loader delegating lookups upward before attempting to load a class itself [1, 23]. This delegation model enforces namespace separation—two classes with the same fully qualified name loaded by different loaders are treated as distinct types, which prevents a malicious class from silently impersonating a system class [1,23]. Loaded classes are also associated with a protection domain that records the code source and the set of permissions granted to it, which is what historically allowed the platform to apply different trust levels to code from different origins [1, 3].

2.4Security Manager*:* The Security Manager was historically the component responsible for enforcing fine-grained, permission-based access control at run time—for example, restricting file-system access, network sockets, or the ability to load native code, based on a configurable security policy [1]. Each security-sensitive operation in the JDK was instrumented with a permission check that consulted the calling code’s protection domain through an access-control stack walk. While effective in principle, the model proved costly to maintain—over a thousand methods in the JDK required permission checks—and it was rarely adopted for securing server-side applications, where it was mostly used, if at all, to harden client-side applets and Java Web Start applications. As a result, Oracle deprecated the Security Manager for removal in Java 17 under JEP 411, citing maintenance burden and limited industry adoption as the primary reasons [17], and subsequently disabled it by default through JEP 486 [18,26]. This change is discussed further in Section 6.

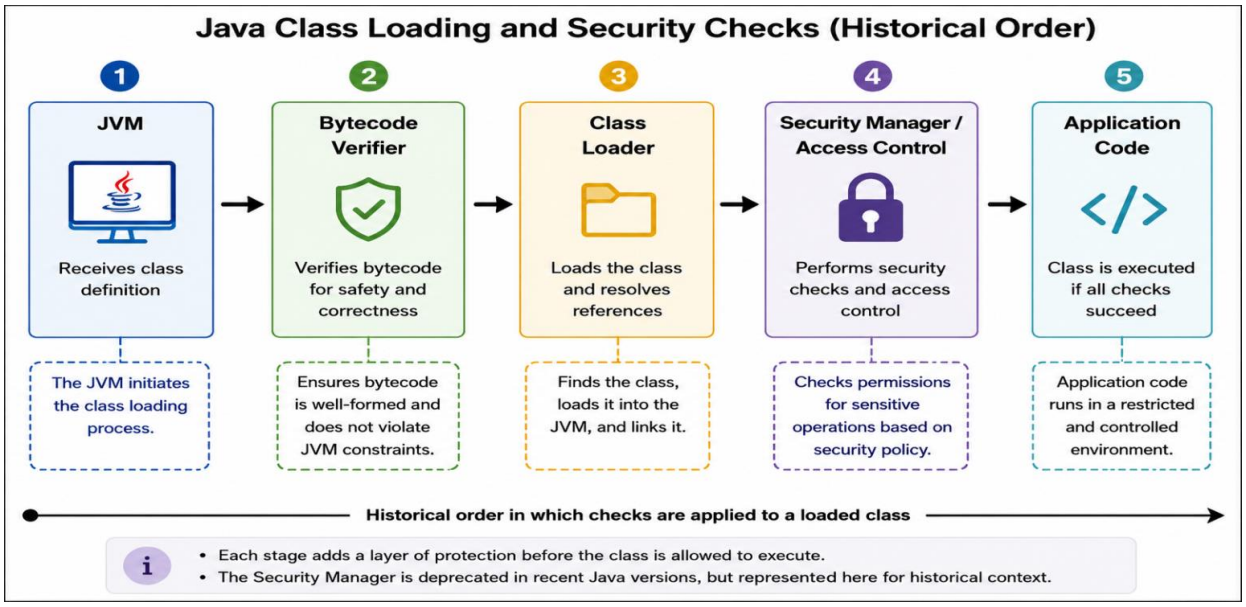

Fig.1: Java class loading and Security Checks

III.JAVA CRYPTOGRAPHY AND SECURE COMMUNICATION

3.1 Java Cryptography Architecture (JCA): The JCA is the framework through which the Java platform exposes cryptographic services—message digests, digital signatures, ciphers, key generators, and secure random-number generation—through a provider-based abstraction [4]. Application code is written against stable interfaces such as Cipher, Message Digest, and Key Pair Generator, while the actual algorithm implementation is supplied by one or more registered providers. This separation allows an application to switch cryptographic implementations, for example to a hardware security module or a FIPS-validated provider, without changing application code [4, 27].

3.2AES: The Advanced Encryption Standard is the symmetric block cipher recommended by NIST for protecting sensitive data, supporting 128-, 192-, and 256-bit keys over a 128-bit block [5]. In Java, AES is accessed through the Cipher class via the JCA and is recommended over legacy symmetric ciphers such as DES and Triple-DES, which are considered cryptographically weak for new designs [4, 5,28].

3.3RSA: RSA is an asymmetric algorithm used for both encryption and digital signatures and is exposed in Java through the Key Pair Generator, Signature, and Cipher classes [4]. RSA underlies much of the public-key infrastructure used by Java's secure-communication stack, including certificate-based authentication in TLS.

3.4SHA-256: SHA-256, part of the SHA-2 family specified by NIST, is the message-digest algorithm used in Java applications for integrity checks, password hashing in combination with adaptive functions, and digital signatures [6]. It supersedes SHA-1 and MD5, both of which are now considered broken for collision resistance and are flagged as a cryptographic weakness—CWE-327, Broken or Risky Cryptographic Algorithm—by OWASP [14, 29].

3.5 Secure Random: Predictable randomness is a common root cause of cryptographic failure; OWASP explicitly lists insufficient entropy among the weaknesses that fall under Cryptographic Failures [14]. Java's SecureRandom class is designed to draw from a cryptographically strong source of entropy and should be used in place of java.util.Random whenever a value—a session token, a key, or a nonce—has security significance [4].

3.6SSL/TLS: Secure communication in Java is implemented through the Java Secure Socket Extension (JSSE), which provides SSL Socket and SSLEngine implementations of the TLS protocol on top of the JCA's key- and certificate-management facilities [4]. Misconfiguration at this layer—accepting all certificates, disabling hostname verification, or permitting deprecated protocol versions—is a frequently cited source of real-world man-in-the-middle vulnerabilities, particularly in micro service and cloud-native deployments where service-to-service traffic must be encrypted by default [22].

Primitive	JCA Class	Typical Use
AES	Cipher	Symmetric encryption of data at rest / in transit
RSA	Key Pair Generator, Signature	Key exchange, digital signatures
SHA-256	Message Digest	Integrity checks, hashing
Secure Random	Secure Random	Keys, tokens, nonces

Table 1:-Summary of Java Cryptographic Primitives

IV.COMMON VULNERABILITIES IN JAVA

Despite the platform-level protections summarized above, application-level vulnerabilities remain common in Java software, largely because they arise from how developers use APIs rather than from flaws in the platform itself. This section discusses five vulnerability classes that recur frequently in OWASP guidance and in Java-focused security research, along with their causes, impact, and prevention strategies. Table 2 summarizes this discussion.

4.1SQL Injection: SQL injection arises when an application builds a query by concatenating untrusted user input directly into the SQL string, allowing an attacker to alter the query's logical structure [7, 8]. A successful injection can expose, modify, or delete arbitrary database records, and in some configurations can be escalated further through database-specific features. The standard remedy is to use parameterized queries through JDBC's PreparedStatement, which separates the SQL command from user-supplied data so that input is always treated as a literal value rather than executable SQL [7, 24]. An empirical study of open-source Java projects found that an automated prepared-statement replacement algorithm correctly removed 94% of identified SQL injection vulnerabilities, demonstrating both the prevalence of the problem and the practical effectiveness of this single mitigation [7]. OWASP's SQL Injection Prevention Cheat Sheet additionally recommends input validation and the principle of least privilege on database accounts as defense-in-depth measures [8].

4.2Deserialization Attacks: Java's native serialization mechanism reconstructs an entire object graph from a byte stream, invoking constructors and methods such as readObject without restricting the set of classes that may be instantiated; if an attacker controls the byte stream, pre-existing classes already present on the classpath can be chained together into a "gadget chain" that executes arbitrary logic [9,10,25]. Because deserialization typically happens before any application-level validation can run, successful exploitation frequently leads directly to remote code execution; one widely cited study identified roughly 40 disclosed CVEs related to Java deserialization between 2018 and 2020 alone, most of which were rated critical [9]. OWASP groups this issue under Software and Data Integrity Failures in its current Top Ten [11]. The most effective mitigation is to avoid deserializing untrusted data using native Java serialization altogether, preferring data formats such as JSON with a strict schema. Where native serialization cannot be avoided, JEP 290 introduced a filtering mechanism that lets an application define an allow-list of classes that may be deserialized, rejecting everything else before object construction completes [16].

4.3Reflection Abuse: The Reflection API allows code to inspect and invoke classes, methods, and fields at run time, including private members when accessibility checks are explicitly disabled through setAccessible(true); because reflective calls are validated against the caller at the point of use rather than enforced by the compiler, careless or malicious use can bypass the language's normal access-control rules [12], [13], [26]. Reflection misuse has historically enabled full sandbox-bypass attacks in client-side Java and continues to be flagged by OWASP as a path to unintended control flow and limited code-injection vulnerabilities [12]. SEI CERT's secure coding standard for Java recommends avoiding reflection to increase the accessibility of classes, methods, or fields wherever an alternative exists, and exercising strict caution when it cannot be avoided [13]. The Java Platform Module System adds a further layer of protection by allowing modules to strongly encapsulate their internal packages, so that reflective access to non-exported types is denied by the module boundary itself rather than relying solely on a runtime security check [19].

4.4Hardcoded Credentials: Developers sometimes embed passwords, API keys, or cryptographic keys directly in source code for convenience during development, a pattern formally cataloged as CWE-798, Use of Hard-coded Credentials [15]. Because source code is frequently stored in version-control history, packaged into compiled artifacts, or shared across teams, hardcoded secrets are easily recovered through decompilation or repository inspection, and once exposed they remain valid until manually rotated [15]. OWASP's Top Ten places this pattern under both Cryptographic Failures and Identification and Authentication Failures, reflecting its dual role in both categories [14]. Secrets should instead be externalized to environment variables, configuration management systems, or dedicated secret-management services, with automated secret-scanning integrated into the build pipeline to catch accidental commits before they reach a shared repository.

4.5Weak Cryptographic Algorithms*:* Legacy code frequently continues to use algorithms that are now considered broken or insufficiently strong—MD5 and SHA-1 for hashing, DES for encryption, or AES used in ECB mode with no proper initialization vector—a pattern OWASP catalogs as CWE-327, Broken or Risky Cryptographic Algorithm, under its Cryptographic Failures category [14]. Weak hashing allows password databases to be cracked efficiently using precomputed rainbow tables, while weak symmetric-cipher configurations can leak structural information about the underlying plaintext. Applications should standardize on NIST-recommended primitives—AES in an authenticated mode such as GCM, and SHA-256 or stronger for hashing—and should use adaptive, salted password-hashing functions such as bcrypt, PBKDF2, or Argon2 rather than a general-purpose hash function for credential storage [5,6,14].

Vulnerability	Primary Cause	Key Mitigation
SQL Injection	Unsanitized input concatenated into SQL	Prepared statements [7], [8]
Insecure Deserialization	Unrestricted object reconstruction from untrusted data	Avoid native serialization; JEP 290 filters [9], [16]

Reflection Abuse	Bypassing compile-time access checks	Avoid where possible; module encapsulation [12], [13], [19]
Hardcoded Credentials	Secrets embedded in source code	Externalize secrets; secret scanning [15]
Weak Cryptography	Use of broken / legacy algorithms	AES-GCM, SHA-256+, salted hashing [5], [6], [14]

Table 2: Common Java Vulnerability Classes

V.SECURE CODING PRACTICES AND MITIGATION TECHNIQUES

Beyond addressing individual vulnerability classes, several broader practices consistently reduce the security risk of Java applications across their lifecycle.

5.1Prepared Statement: As discussed in Section IV-A, parameterized queries through PreparedStatement should be the default mechanism for any database access that incorporates user-supplied input, with safely implemented stored procedures as an acceptable alternative [7, 8].

5.2Input Validation: All input that crosses a trust boundary—from a user, an external API, or a file—should be validated against an explicit allow-list of expected formats wherever possible, rather than relying solely on blacklisting known-bad patterns, since blacklists are inherently incomplete [8].

5.3Principle of Least Privilege: Components, database accounts, and operating-system processes should be granted only the minimum set of permissions required to perform their function. This principle, originally enforced in Java through the Security Manager's permission model, remains a foundational defence even as that specific mechanism is removed, and is increasingly enforced instead through container-level and orchestration-level access controls in cloud-native deployments [1, 17, 22].

5.4Secure Dependency Management: Modern Java applications depend on large numbers of third-party libraries pulled in transitively through build tools such as Maven and Gradle, and vulnerability in any one of them becomes vulnerability in the application that depends on it. Software Composition Analysis tools that continuously scan a project's dependency tree against known-vulnerability databases—such as the OWASP Dependency-Check project—are now considered a standard part of a secure software development lifecycle [22].

5.5 Regular Updates: Because new vulnerabilities are continuously discovered in both the JDK itself and in commonly used libraries, applying security patches promptly is one of the simplest and most effective mitigations available; this is reflected in the fact that many disclosed Java deserialization vulnerabilities were addressed not by an application-level code fix but by an upstream patch or workaround that organizations had to actively apply [9].

VI.RECENT DEVELOPMENTS AND FUTURE SCOPE

6.1Security Manager Deprecation*:* As discussed in Section 2.4, JEP 411 deprecated the Security Manager for removal starting in Java 17, citing its limited adoption on the server side, the high maintenance cost of the permission checks scattered across the JDK, and the availability of better-suited alternatives such as containers and hypervisors for isolating an entire Java runtime from sensitive resources [17]. JEP 486 subsequently disabled the Security Manager by default, marking the practical end of the permission-based access-control model that defined Java security for over two decades [18].

6.2Module System: The Java Platform Module System, delivered in Java 9 under JEP 261 (Project Jigsaw), introduced strong encapsulation as a language-level and JVM-level concept rather than a convention enforced only by documentation [19]. Modules can choose to export only specific packages and, by default, deny reflective access to internal implementation details, which directly mitigates the reflection-abuse vulnerability class discussed in Section IV-C and is intended to take over part of the protective role previously played by the Security Manager [19].

6.3Cloud-Native Java Security: As Java applications increasingly run as containerized microservices orchestrated by platforms such as Kubernetes, the security perimeter has shifted away from in-process access control toward infrastructure-level mechanisms: network segmentation, service meshes that provide mutual TLS between services, and zero-trust architectures that re-verify every request rather than trusting traffic by network location [22]. A recent survey of cloud-native security identifies container and micro service security, orchestration-platform security, and Develops practices as the areas requiring the most attention as Java workloads migrate to these environments [22].

6.4AI-Based Threat Detection: Machine learning and deep learning techniques are increasingly being applied to automated vulnerability detection in source code, including Java-specific gadget-chain discovery for deserialization vulnerabilities [9]. However, a systematic evaluation of deep-learning-based vulnerability detectors found that their accuracy can drop by more than 50% when moved from controlled research datasets to realistic, real-world prediction settings, due to issues such as data duplication and unrealistic class distributions in training data [21]. A broader survey of machine-learning-based vulnerability detection similarly concludes that, while the field has made measurable progress, current techniques are not yet a substitute for established static analysis and secure coding practices, and are best understood as a complementary layer in a defence-in-depth strategy [20].

VII.CONCLUSION

Java has traditionally relied on a comprehensive security framework that combines multiple protective mechanisms rather than depending on a single security control. Core components such as the Java Virtual Machine (JVM), bytecode verification process, class-loading system, and the Security Manager have collectively contributed to create a secure execution environment. In addition, the Java Cryptography Architecture (JCA) provides essential cryptographic services that support the protection of sensitive information during storage and transmission. The review indicates that many security issues observed in Java applications are not primarily caused by weaknesses in the platform itself. Instead, vulnerabilities often arise from improper implementation practices and the incorrect use of application programming interfaces (APIs). Technologies such as JDBC, Java serialization, reflection mechanisms, and cryptographic libraries can introduce security risks when they are not used according to established guidelines. Consequently, vulnerabilities such as SQL injection, insecure deserialization, exposure of credentials, and cryptographic misconfigurations continue to be reported in both industry and academic studies. A significant observation is that most of these security challenges are well understood and can be mitigated through established secure-development practices. Techniques including parameterized queries, strict input validation, controlled deserialization, proper access control, secure secret management, and the adoption of strong cryptographic standards can substantially reduce the attack surface of Java applications. Therefore, developer awareness and adherence to secure coding principles remain critical factors in building resilient software systems. As the Java ecosystem continues to evolve, its security landscape is also undergoing significant changes. The gradual retirement of the Security Manager, the increasing adoption of the Java Platform Module System (JPMS), the widespread use of containerized and cloud-native environments, and the emergence of artificial intelligence for vulnerability analysis is reshaping how application security is implemented and maintained. In this changing environment, secure software development can no longer rely solely on platform-provided safeguards. Instead, a proactive approach that integrates security throughout the software development lifecycle is essential for developing trustworthy and robust Java applications.

REFERENCES

1. L. Gong, G. Ellison, and M. Dageforde, Inside Java 2 Platform Security: Architecture, API Design, and Implementation, 2nd ed. Boston, MA: Addison-Wesley, 2003.
2. X. Leroy, “Java bytecode verification: Algorithms and formalizations,” Journal of Automated Reasoning, vol. 30, no. 3–4, pp. 235–269, 2003.
3. L. Gong, “Java security: A ten-year retrospective,” in Proc. Annual Computer Security Applications Conference (ACSAC), Honolulu, HI, USA, 2009.
4. Oracle Corporation, “Java Cryptography Architecture (JCA) Reference Guide,” Java SE Documentation. [Online]. Available: https://docs.oracle.com/en/java/javase/11/security/java-cryptography-architecture-jca-reference-guide.html
5. National Institute of Standards and Technology, Advanced Encryption Standard (AES), Federal Information Processing Standards Publication FIPS 197, Nov. 2001 (updated 2023).
6. National Institute of Standards and Technology, Secure Hash Standard (SHS), Federal Information Processing Standards Publication FIPS 180-4, Aug. 2015.
7. S. Thomas, L. Williams, and T. Xie, “On automated prepared statement generation to remove SQL injection vulnerabilities,” Information and Software Technology, vol. 51, no. 3, pp. 589–598, 2009.
8. OWASP Foundation, “SQL Injection Prevention Cheat Sheet,” OWASP Cheat Sheet Series. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html
9. I. Sayar, A. Bartel, E. Bodden, and Y. Le Traon, “An in-depth study of Java deserialization remote-code execution exploits and vulnerabilities,” ACM Trans. Softw. Eng. Methodol., vol. 32, no. 1, Art. 25, pp. 1–45, 2023.
10. S. Rasheed, J. Dietrich, and A. Tahir, “A hybrid analysis to detect Java serialisation vulnerabilities,” in Proc. 35th IEEE/ACM Int. Conf. Automated Software Engineering (ASE), 2020, pp. 1209–1213.
11. OWASP Foundation, “A08:2021 – Software and Data Integrity Failures,” OWASP Top 10:2021. [Online]. Available: https://owasp.org/Top10/2021/A08_2021-Software_and_Data_Integrity_Failures/
12. OWASP Foundation, “Unsafe Use of Reflection.” [Online]. Available: https://owasp.org/www-community/vulnerabilities/Unsafe_use_of_Reflection.
13. SEI CERT, “SEC05-J. Do not use reflection to increase accessibility of classes, methods, or fields,” SEI CERT Oracle Coding Standard for Java, Carnegie Mellon University.
14. OWASP Foundation, “A02:2021 – Cryptographic Failures,” OWASP Top 10:2021. [Online]. Available: https://owasp.org/Top10/2021/A02_2021-Cryptographic_Failures/
15. MITRE Corporation, “CWE-798: Use of Hard-coded Credentials,” Common Weakness Enumeration. [Online]. Available: https://cwe.mitre.org/data/definitions/798.html
16. R. Riggs, JEP 290: Filter Incoming Serialization Data, OpenJDK, 2016. [Online]. Available: https://openjdk.org/jeps/290
17. S. Mullan, L. Andersen, and W. Wang, JEP 411: Deprecate the Security Manager for Removal, OpenJDK, 2021. [Online]. Available: https://openjdk.org/jeps/411
18. OpenJDK, JEP 486: Permanently Disable the Security Manager, 2024. [Online]. Available: https://openjdk.org/jeps/486
19. M. Reinhold, JEP 261: Module System, OpenJDK, 2017. [Online]. Available: https://openjdk.org/jeps/261
20. T. Marjanov, I. Pashchenko, and F. Massacci, “Machine learning for source code vulnerability detection: What works and what isn’t there yet,” IEEE Security & Privacy, vol. 20, no. 5, pp. 60–76, 2022.
21. S. Chakraborty, R. Krishna, Y. Ding, and B. Ray, “Deep learning based vulnerability detection: Are we there yet?,” IEEE Trans. Softw. Eng., vol. 48, no. 9, pp. 3280–3296, 2022.
22. T. Theodoropoulos et al., “Security in cloud-native services: A survey,” Journal of Cybersecurity and Privacy, vol. 3, no. 4, pp. 758–793, 2023.
23. Baeldung, “Class Loaders in Java.” [Online]. Available: https://www.baeldung.com/java-classloaders

24. GeeksforGeeks, “How to Handle SQL Injection in JDBC Using PreparedStatement?” [Online]. Available: https://www.geeksforgeeks.org/java/how-to-handle-sql-injection-in-jdbc-using-preparedstatement/
25. Baeldung, “Deserialization Vulnerabilities in Java.” [Online]. Available: https://www.baeldung.com/java-deserialization-vulnerabilities
26. Baeldung, “Is Java Reflection Bad Practice?” [Online]. Available: https://www.baeldung.com/java-reflection-benefits-drawbacks
27. Alam, M. I. (2020). Enhancing cloud security using multi-Level dna cryptography. *Splint International Journal of Professionals*, *7*(1), 75-82.
28. Irfan Alam, M., & Singh, S. N. (2021). Designing and implementing cloud security using multi-layer DNA cryptography in python. *Trends in Wireless Communication and Information Security: Proceedings of EWCIS 2020*, 375-385.
29. Alam, Md Irfan. "Enhancing cloud security using multi-Level dna cryptography." *Splint International Journal of Professionals* 7.1 (2020): 75-82.