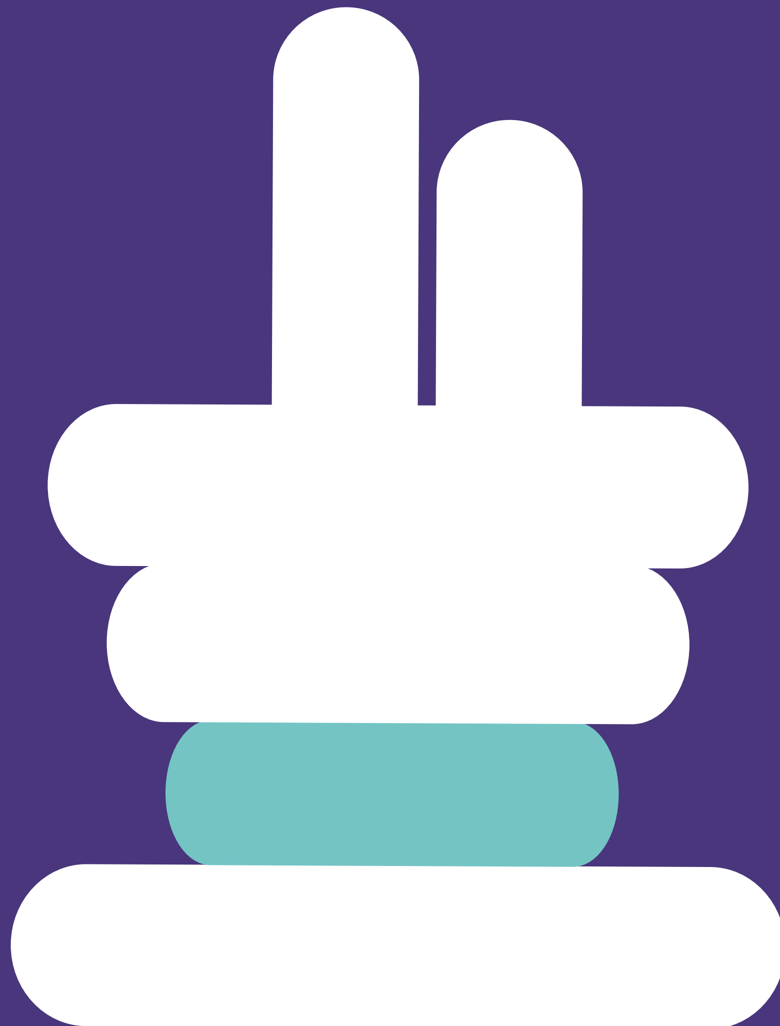
A Security Guide for

Java Developers



kiuwan
An Idera, Inc. Company

INDEX

Overview..1

Addressing Security When Coding in Java................2

Java Risk: Insecure Deserialization.........................3

Java Risk: Insufficient Authorization and Authentication..4

Java Risk: Using Broken Cryptography in the API..5

Java Risk: Unsafe Reflection..................................7

Java Risk: Injection flaws.......................................8

The Last Word in Java Security.............................10

Overview

The Java programming language is **versatile and powerful**. It can be used in a wide variety of settings to allow developers to create robust, high-performance applications — from small devices all the way up to large enterprise systems. Despite its age, Java remains one of the most popular programming languages in use today. Developers appreciate its stability, ease of development, and vast ecosystem of libraries and tools.

When **Java** came out in the mid-1990s, it promised to revolutionize programming languages at a time when a lot of business programming was done in C or C++. People who have experienced those languages in a professional setting know that they can be full of potential problems that at times can be hard to detect. Java basically removed many of those pitfalls. For example, instead of worrying about allocating and freeing memory manually, Java does all of that for you. This eliminates a whole class of vulnerabilities, and as a result, Java has gained a reputation as being more secure than other programming languages. However, just because Java is considered more secure overall doesn't mean that every piece of code written in Java is automatically secure.

This Java security guide will help organizations prepare effective approaches for securely developing in Java language. The guide highlights specific risks Java developers face and demonstrates what proactive steps can be taken to protect these applications. This guide will explore the following topics and top common risks:

- **Addressing Security When Coding in Java**
- Java Security Risk: **Insecure Deserialization**
- Java Security Risk: **Insufficient Authorization and Authentication**
- Java Security Risk: **Broken Cryptography**
- Java Security Risk: **Unsafe Reflection**
- Java Security Risk: **Injection Flaws**

Addressing Security When Coding in Java

There are several well-known **security vulnerabilities in the Java** programming language. One is the "remote code execution" vulnerability, which allows an attacker to execute arbitrary code on a victim's machine by exploiting a flaw in how Java handles deserialization of objects. This can be used to remotely take control of a machine or install malware.

Another common vulnerability is the "man-in-the-middle" attack, where an attacker intercepts communications between two parties and impersonates one or both of them to gain access to sensitive information.

Finally, there have been numerous cases where attackers have exploited flaws in the way that Java implements SSL/TLS encryption to eavesdrop on communications or even inject malicious content into what appears to be a secure connection.

While these vulnerabilities and attacks may not be unique to Java — indeed, most attack vectors and patterns can be broadly categorized into similar classifications or taxonomies — the way they are executed and defended against specifically in Java will differ from other programming languages. Security risks in Java can be potentially costly for organizations if left unaddressed. Consider the cost of each of these risks:

- **Malware infections:** Java-based malware is becoming increasingly prevalent and can easily infect systems that have not been properly secured. Once infected, these systems may be used to launch attacks against other computers on the network or to steal sensitive data. Some estimates put the cost of a malware attack at $2.5 million.
- **Denial of service attacks:** Unsecured Java applications can be exploited by attackers to launch denial of service (DoS) attacks, which can cause significant disruptions and cost businesses dearly in terms of lost productivity and revenue. Some reports indicate that DoS attacks cost $22,000 for every minute of downtime.
- **Data security breaches:** Poorly secured Java apps may also provide an easy way for hackers to gain access to sensitive corporate data stored on the server or in databases connected to the application. This could lead to serious financial losses as well as reputational damage for the affected organization. While data breach damages are much harder to put an average to in a way that is innately classifiable and meaningful, there are some estimates placing the average costs associated with data breaches at $4.54 million when coupled with ransomware.

Neglecting application security for Java can present very real potential consequences for your company.

Java Security Risk: *Insecure Deserialization*

Insecure deserialization is a Java security risk that can occur when untrusted data is deserialized by a Java application. This could lead to malicious code execution or denial of service attacks. Insecure deserialization can be exploited if an attacker can supply malicious data to the application — for example, by modifying packets sent to the application over the network or by uploading a malicious file to the server.

In the Java programming language, insecure deserialization is most commonly exploited through user input, such as with POST parameters or cookies. Attackers can supply maliciously crafted data that will be executed when deserialized. This can allow them to run arbitrary code on the server, leading to complete compromise of the system. In some cases, attackers may simply exploit this flaw for DoS purposes by causing the application to crash or hang indefinitely.

User input isn't the only way an attacker can exploit insecure deserialization. They can also target serialized objects stored in files or databases. If these objects are not properly validated before being unserialized, an attacker could modify them in a way that allows them to execute arbitrary code when loaded by the application. Additionally, if an application log contains serialized objects, an attacker could tamper with it to insert malicious data that would be executed when viewed later.

Exploiting insecure deserialization usually requires some knowledge of how the targeted application works and what kind of data it stores internally (e.g., object types and structure). However, tools are available that automate this process by generating payloads specifically designed to attack common vulnerabilities like those described above.

To mitigate this risk, it is important to only deserialize data from trusted sources and/or use a platform that has built-in protections against insecure deserialization vulnerabilities. When data is deserialized, it is converted back into an object from its serialized form. This process can be exploited if the data comes from an untrusted source, as malicious code could be executed when the object is reconstituted. Thus, it is important to only deserialize data that comes from a trusted source. Using a platform with built-in protections against insecure deserialization vulnerabilities can further mitigate this risk.

Java Security Risk: *Insufficient Authorization and Authentication*

One of the most common software security risks associated with Java development is insufficient authorization and authentication. This can occur when developers fail to properly check that a user or process has the correct permissions before allowing access to sensitive data or resources.

In some cases, this can be due to a misunderstanding of how Java's security model works. The Java security model is based on a "sandbox" concept where untrusted code running in one part of the system is prevented from accessing other parts of the system that it should not have access to. However, if developers do not set up the sandbox correctly, it may be possible for malicious code to escape the sandbox and gain access to sensitive data.

In other cases, insufficient authorization and authentication checks may be due simply due to carelessness or rushing through the implementation phase without thoroughly testing all possible scenarios. Either way, failing to properly check permissions leaves your application vulnerable to attack.

There are *two main ways* that you can mitigate this risk:

1. Properly implemented authentication checks

These should include confirming user authorization before granting them access to sensitive data or resources. This can be done through methods like password hashing and checking against a database of known users.

In Java, password hashing is a process where the user's password is run through a one-way mathematical function. The outcome of the function is then stored in the database instead of the actual password. When a user attempts to log in, their inputted password is run through the same hash function. If the output of both functions matches, the authentication is successful and access can be granted.

There are many ways to hash passwords, but some common methods include SHA-256 and bcrypt. It's important to note that when storing hashed passwords, you should never use plain text or reversible encryption. Doing so would defeat the purpose of hashing.

Another way to mitigate insufficient authorization and authentication risks in Java is by verifying the identity of users before granting them access to sensitive data or resources. This step helps ensure that only authorized individuals can view or modify information. There are several ways this check can be performed, including prompting for additional identifying information (e.g., birth date or Social Security number), using two-factor authentication (requiring both something known and something possessed), and checking against biometric data (e.g., a fingerprint).

2. Properly implemented authorization checks

Once authenticated, these checks ensure that a user only has access to those items they are authorized for. This often takes the form of role-based permissions where each user is only permitted certain actions based on their assigned role.

For example, a common way to implement authorization checks is through Java's Access Control Lists (ACLs). ACLs are used to define which users or groups have access to which resources. They can also be used to specify the type of access (e.g., read, write, execute) a particular user or group has. By properly configuring ACLs, you can help ensure that users have access to only those resources they are authorized for.

Another way you can mitigate the risk of insufficient authorization and authentication in Java is by using security policies. Security policies typically contain instructions on how your application should handle various security-related tasks such as authenticating users, authorizing access control decisions, and encrypting data in transit. With clear security policies that are followed strictly throughout your development process, everyone involved in building and maintaining your application will know exactly what needs to be done to maintain security.

Java Security Risk: *Using Broken Cryptography in the API*

Java's cryptographic API offers a wide range of capabilities, from authenticating users to encrypting data. However, this flexibility comes with considerable complexity, which can lead to security vulnerabilities if not used properly. Weak cryptography algorithms, improper key management, and insecure implementation of cryptographic functions can all lead to security vulnerabilities in Java applications:

Weak Cryptography

One of the most common mistakes when coding in Java is using weak cryptography algorithms. For example, the Data Encryption Standard (DES) has been broken for two decades and should no longer be used. Other algorithms, such as the Advanced Encryption Standard (AES), are still secure but may become vulnerable over time as computing power increases.

DES was broken in the late 1990s by brute force attacks with specialized hardware. DES has a key size of only 56 bits, which is too small to resist modern attack methods. AES was designed to be a replacement for DES and uses much larger key sizes by default (128, 192, or 256 bits). AES is also **resistant to known attacks as of 2019** but may eventually become vulnerable as computers continue to get more powerful.

It's important to use strong cryptography when developing Java applications. Weak algorithms can lead to data being compromised by attackers who can break them. When choosing an algorithm, it's important to consider not only its strength today but also its potential future vulnerabilities so that you can select one that will remain secure for years to come.

Improper Key Management

As seen time and time again, data breaches are becoming more prevalent with each passing year. And while many factors contribute to these types of incidents, one of the most common is poor key management. When it comes to cryptography, the keys used play a critical role in keeping data safe. As such, they must be properly generated, stored, and rotated regularly. Unfortunately, this is often where things go wrong.

For example, let's say you have a key that you use to encrypt sensitive data. If this key is compromised (e.g., someone steals it from your database), then all of your data can be accessed by the attacker. This is why proper key management is so important; if done correctly, it can help to mitigate the risks associated with data breaches and other security incidents.

Various cryptography key management best practices should be followed to reduce the risk of keys being compromised. As mentioned earlier, one best practice is to generate keys using a strong random number generator (RNG) instead of using predictable or easily guessed values. Additionally, it's important to rotate keys regularly so that even if an attacker manages to obtain one key, it will only be valid for a limited period. Other measures can be taken as well, such as storing keys in hardware security modules (HSMs) which provide additional protection against physical attacks.

Insecure Implementation

The Java Cryptographic API offers a variety of ways to perform cryptographic operations. This can be beneficial because it gives developers flexibility in how they implement cryptography. However, it can also be difficult to know if you are using the best method for your particular use case.

One way to generate random numbers in Java is with the **java.security.SecureRandom class.** This class uses a strong source of entropy, such as /dev/urandom on Linux systems, so that the output is truly random and secure. To initialize the SecureRandom object properly, you need to seed it with data from a reliable source of entropy like /dev/urandom.

Another issue with Java's cryptographic API is that some methods may be less secure than others or more suited for a particular use case. For example, there are multiple ways to generate random numbers in Java and each has its own set of trade-offs:

- Java Math Random
- Secure Random
- ThreadLocalRandoom
- SplittableRandom

Which one should you choose? It depends on your application requirements. If you need crypto-graphically strong values then go for SecureRandom. For applications where runtime performance is not as important, SplittableRandom may be a better option.

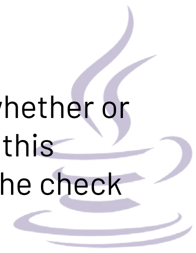
Java Security Risk: *Unsafe Reflection*

The Java reflection API allows developers to introspect and manipulate the runtime behavior of applications. While this can be a very powerful tool, it can also be dangerous if not used properly. In particular, allowing untrusted input into the reflection API can enable attackers to bypass security controls or execute arbitrary code on a target system.

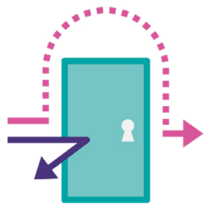
There are two primary ways that attackers can exploit vulnerable code using reflection:

Bypassing security checks

By carefully crafting their input, an attacker may be able to cause a program to skip critical security checks that would normally prevent them from accessing sensitive data or functionality. This could allow an attacker with low privileges to gain access to high-privilege data or functionality, among others.



For example, let's say there is a Java program that has a security check in place that verifies whether or not the user has permission to view sensitive data. An attacker could use reflection to bypass this security check by carefully crafting their input in such a way that causes the program to skip the check and grant them access to the sensitive data.



This type of attack can be difficult to detect and defend against because it exploits vulnerabilities in the code itself rather than anything on the system level (such as permissions). As such, it's important for developers who are writing code that will use reflection to ensure that all possible inputs are properly validated before processing them.

Executing arbitrary code

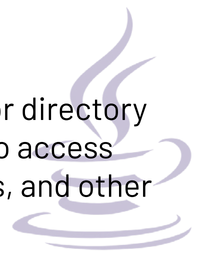
Attackers may also attempt to use reflection to dynamically load and execute malicious code on a target system. This type of attack is particularly dangerous because it can be used to bypass security controls such as firewalls or intrusion detection systems (IDS).

Attackers can use reflective invocations from within Classes loaded with different class loaders, for example, and invoke private methods from within different classes. Successful attempts to exploit vulnerable Java code using reflection can lead to attacks that then target other vulnerabilities, such as insecure deserialization

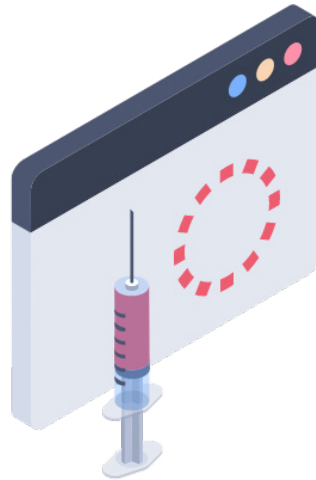
Java Security Risk: *Injection flaws*

One of the most common security risks in Java development is injection flaws. Injection flaws, such as SQL injection or LDAP injection, occur when untrusted user input is inserted into a command or query that is then executed by the application. This can allow attackers to gain access to sensitive data, modify data, or even execute malicious code on the server.

To prevent SQL injection attacks, it is important to validate and sanitize all user input before inserting it into a command or query. For example, when building SQL queries, use parameterized queries instead of concatenating strings together. Additionally, make sure you are using an up-to-date database driver that supports parameters.



The Lightweight Directory Access Protocol (LDAP) is an open, cross-platform protocol used for directory service authentication. LDAP provides a communication language that applications can use to access directory servers that often store information such as usernames, passwords, account details, and other data that is then shared within the network.



LDAP injections happen when an application inserts unsanitized inputs directly into an LDAP statement. When this occurs, the attacker can use control characters in the input to modify the LDAP filter syntax causing the server to execute unintended queries or unintentionally disclose data stored in fields accessible via query results (e.g execution of logical operators such as OR or !). The easiest way to prevent LDAP injection is to ensure that user input does not contain any special characters used in LDAP filters, such as ( ) * & | !

If user input must contain one of these characters, then consider using whitespace as a delimiter between multiple terms entered by users e.g. "a b c" instead of "a&b|c".

It's best to preemptively look into each specific type of injection flaw and its own set of best practices for prevention.

The Last Word in Java *Security*

This Java security guide provides foundational data that can inform and enhance the security around your application development policy. In addition to adopting best practices and keeping up-to-date with security vulnerabilities related to Java and its dependencies, it's ideal to partner with a code security solution such as Kiuwan to bolster your mobile and web application development security. Kiuwan secures applications with code security and analysis tools by arming you with the capability to automatically identify and remediate security vulnerabilities.

Kiuwan provides static application security testing (SAST) through Kiuwan Code Security, which is compliant with standards such as OWASP and Common Weakness Enumeration (CWE). Additionally, Kiuwan offers software composition analysis (SCA), called Kiuwan Insights Open Source. This SCA reduces risk from third-party components, assists in addressing vulnerabilities, and ensures license compliance as well as policy automation throughout your software development life cycle.

Java applications can be made more secure with Kiuwan. In addition to Java, Kiuwan's security capabilities cover more than 30 coding languages. **Schedule a demo** with Kiuwan today.

YOU KNOW ***CODE,*** *WE KNOW* ***SECURITY!***

GET IN TOUCH:

Headquarters

2950 N Loop Freeway W, Ste 700

Houston, TX 77092, USA

United States **+1 732 895 9870**

Asia-Pacific, Europe, Middle East and Africa **+44 1628 684407**

contact@kiuwan.com

Partnerships: **partners@kiuwan.com**