WARATEK

WHITE PAPER · NOV 2026

JAVA DESERIALIZATION · CWE-502

Dynamic Micro-compartmentalization: The Solution to Deserialization Vulnerabilities

Background on the Java deserialization vulnerability, how the JDK, the attacker toolchain and AI have reshaped the problem over the past decade, the limitations of existing mitigation techniques, and why Dynamic Micro-compartmentalization and Security-as-Code remain a uniquely effective solution to this vulnerability class.

AUTHOR Apostolos Giannakidis, CTO

PUBLISHED November 5, 2026

TOPIC Runtime protection for Java deserialization

1,339	**14 vs 10**	**-7 days**	**45+**
Deserialization CVEs in the first ten months of 2026	Spring deserialization CVEs in 2026 vs past 12 years	Mean time-to-exploit (Mandiant M-Trends 2026)	Java SE Deserialization vulnerabilities fixed since 2015

Java deserialization has been a top-tier enterprise security risk for more than a decade, and AI is now accelerating it. Attackers can find new ways to exploit it faster than vendors can patch or security teams can update their filters. This paper explains why the traditional defenses keep falling behind and what that means for organizations running Java at scale. Read how Waratek's runtime micro-compartmentalization closes the gap without code changes or ongoing tuning of filters.

01 The Deserialization Problem

Serialization is the process that converts an in-memory object into a stream of bytes to store it in the filesystem or transfer it to another remote application. Deserialization is the reverse process that converts the serialized stream of bytes back to an object in the machine's memory.

Most programming languages provide facilities to perform native serialization and deserialization, and most of them are vulnerable. In Java the native API is java.io.ObjectInputStream.readObject(), but the same class of flaw exists in XML, YAML, Hessian and JSON polymorphic deserializers.

The vulnerability arises when the application deserializes data from untrusted sources. Malicious payloads can dictate which classes the runtime instantiates and with what field values. This is because both the type name and the field values travel inside the serialized, untrusted, data (payload). As each object is rebuilt in memory, the deserializer automatically invokes certain class methods (callbacks) before the application has any chance to inspect or validate the resulting object. By chaining these callbacks across classes already present in the classpath, known as gadgets, an attacker turns ordinary (non-malicious) code into a path to dangerous operations such as remote code and command execution (RCE), unsafe reflection, filesystem writes, JNDI lookups or resource exhaustion.

Despite a decade after it was first weaponized at scale and years of mitigation efforts, unsafe deserialization remains one of the most dangerous weaknesses in enterprise software. CWE-502 (Deserialization of Untrusted Data) entered the OWASP Top 10 in 2017, and in 2025 it climbed to rank 15 in the CWE Top 25.

In the last couple of years alone, native Java deserialization was the entry point for actively exploited attacks on popular open source software, such as Apache Tomcat (CVE-2025-24813) and Spring for GraphQL (CVE-2026-59285). Enterprise applications have also had a long history of deserialization flaws, with an increasing rate in 2026. Oracle WebLogic Server alone has fixed at least 34 deserialization CVEs since January 2015. For example, the recent CVE-2026-60198, CVE-2026-60202 and CVE-2026-60204, all CVSS 9.8, allow remote code execution over T3/IIOP, enabling unauthenticated attackers to fully compromise systems running WebLogic. Another high-profile case is CVE-2026-35273, which allowed unauthenticated attackers to perform remote code execution in Oracle PeopleSoft Enterprise PeopleTools. This flaw was exploited by ShinyHunters (UNC6240) and initially targeted global organizations in the education sector, but in a September follow-up, Mandiant reported a renewed, larger wave that spread beyond education to technology, IT services, healthcare, agriculture, transport and government, affecting dozens of systems worldwide.

Meanwhile, Oracle took steps to mitigate the issue by strongly encapsulating the JDK internals and giving defenders context-specific deserialization filters. Oracle also permanently disabled the Security Manager in the JDK and plans to remove it completely. This was a reasonable decision given how complicated the Security Manager was to use in enterprise apps and how demonstrably easy it was to bypass.

Meanwhile, researchers, and now AI tools, keep finding new gadget chains faster than allowlists and blocklists can be updated.

We must highlight that deserialization flaws are not confined to application code, libraries and application servers. The Java runtime platform itself depends deeply on the serialization mechanism. Since January 2015, Oracle's Critical Patch Updates have fixed at least 45 deserialization vulnerabilities in Java SE, in almost every Java SE patch release. All of them allow unauthenticated attackers to exploit the system over the network. These flaws were found across several JDK components, from Serialization to Scripting, JNDI, JAXP, RMI, Libraries and Security. Because they live in the Java runtime itself, they ship inside every JVM running that release, and any application that deserializes untrusted data anywhere in its stack inherits them.

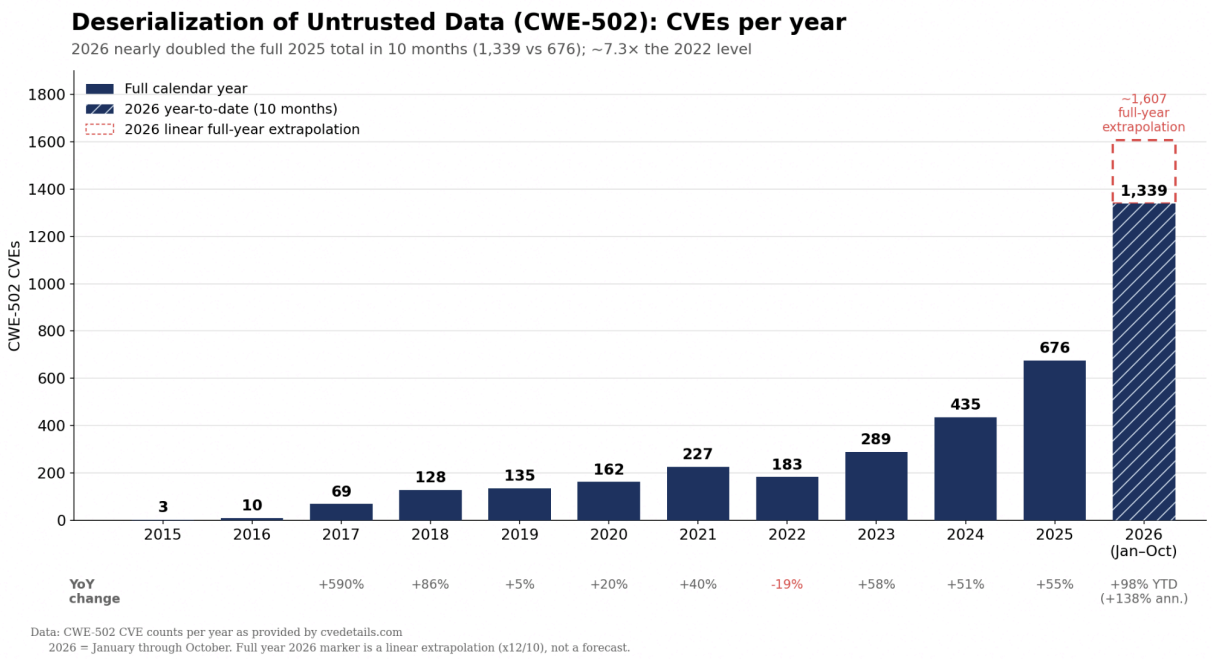
2026: The Deserialization Vulnerability Tsunami

Here's the thing. AI didn't just make finding vulnerabilities faster; it changed the slope of the curve. CVE volume across many vulnerability classes has surged in 2026, which would be remarkable on its own. But deserialization isn't just riding that wave. It's outrunning it, on pace for roughly 2.4 times last year's total.

Let's look at the numbers. In just the first ten months of 2026, 1,339 deserialization CVEs were reported. That's nearly double all of 2025, which had 676, and more than seven times the 2022 level. Extend the line to December 2026 and we'll land at around 1,600 for the year. That's not a trend line. That's a hockey stick.

FIGURE 1 · CWE-502 CVES PER YEAR

2026 nearly doubled the full 2025 total in ten months



This tsunami didn't arrive by accident. It arrived the moment AI started hunting for vulnerabilities. Some of the clearest evidence comes from Spring, the framework that much of enterprise Java and Spring AI are built on.

For years, Spring received six or seven security reports per month. Then in March 2026 it received 55. In April it received 482 across 65 projects, and 370 of those came from its own new AI-powered scanning. This is not a gradual increase; it is roughly seventy times the old monthly rate. Broadcom called the June release that followed the largest set of Spring security updates in the 23 years of the project's history. And it didn't stop there.

On August 20, 2026, Broadcom shipped another 91 CVEs across Spring Framework and related projects in a single batch, affecting nearly 150,000 downstream components. One notable case is the unsafe deserialization CVE-2026-59285 in Spring for GraphQL, whose APIs can already be exposed to AI clients as MCP tools, so the same endpoint that answers an agent's tool call can be the one deserializing an attacker's malicious gadget chains.

We also recently saw an unsafe deserialization vulnerability in LangChain4j (CVE-2026-97869), a widely used Java library for building LLM applications and AI agents. The untrusted data was deserialized when the langchain4j-agentic module restored persisted AI agent state using Jackson default typing, which allowed arbitrary class instantiation. Why is this interesting? Because it reveals a new pattern: **an AI agent's memory is now a deserialization source**. It's a stored-deserialization attack inside agentic AI.

Our analysis of CVE data reveals four key insights that help explain how AI has contributed to the recent surge in reported deserialization CVEs.

01

Discovery was the bottleneck, and AI made it significantly easier

Finding a gadget chain used to require painstaking manual effort, tracing call graphs and building payloads by hand. GadgetHunter (ACM FSE 2026) reports about 23.5 seconds and $0.04 per candidate chain. Using an AI coding agent, Atredis Partners confirmed 17 gadget chains, 6 of them novel, against WebSphere, WildFly and Payara, all hardened against known chains, in just two days.

02

Chain discovery is instant weaponization

For most bug classes, finding a flaw, crafting a payload and developing an exploit are separate steps. In deserialization, a confirmed gadget chain in an application's classpath already is a working exploit for that application. If a chain is found in a shared library, a single discovery can weaponize many applications at once.

03

Gadget chain tools stalled while AI research races ahead

ysoserial, the most popular gadget chain tool, has not merged a new chain since 2021. AI-assisted research is producing new gadget chains and variants faster than ever, including variants that defeat Java's module system and class filter hardening.

04

Patching cannot keep up

Mandiant's M-Trends 2026 puts the mean time to exploit at negative seven days. Effectively, exploitation now starts before a patch is available.

Nearly twelve years after ysoserial and ten years after Oracle first shipped Java deserialization filtering (JEP 290), deserialization is still a prevalent issue, not a legacy problem fading out with old application servers. It is being found faster than ever, in the newest code, by AI tools.

02 A Brief Background

In the Java ecosystem, deserialization was first weaponized at scale in 2015. In January 2015 Chris Frohoff and Gabriel Lawrence presented the now famous *Marshalling Pickles* at AppSec California and released ysoserial, a proof-of-concept tool for generating payloads that exploit unsafe Java object deserialization. Its first release contained four gadget chains for Commons Collections, Groovy and Spring, and it quickly became the de facto proof-of-concept tool for Java deserialization gadget chains. In November 2015 Stephen Breen of FoxGlove Security used ysoserial chains to demonstrate RCE against popular servers including WebLogic, WebSphere, JBoss, Jenkins and OpenNMS.

The most famous gadget was a class in the Apache Commons Collections (ACC) library, the infamous InvokerTransformer. The finding gained traction because of the library's popularity. Any application, server or framework that shipped ACC on its classpath and deserialized untrusted data was most likely exploitable, allowing almost any malicious actor to perform remote code execution. Oracle WebLogic (CVE-2015-4852) and Red Hat JBoss (CVE-2015-7501) were among the first affected products.

What does InvokerTransformer do, and why does it matter?

InvokerTransformer transforms objects by invoking a method on them **through reflection**, with the method name, parameter types and arguments stored in serializable fields. Attackers abuse this to invoke any method they want. Chained with ConstantTransformer and ChainedTransformer, it can call Runtime.getRuntime().exec(cmd) instead of transforming a collection element.

To make this happen, the attacker builds an **object graph** whose deserialization callbacks lead from an entry point to a dangerous sink. Each reusable class or method in that path is called a **gadget**, and the whole path is a **gadget chain**. Deserialization research splits chains into four components:

STEP 1	STEP 2	STEP 3	STEP 4
Entry point	**Trampoline**	**Main gadgets**	**Sink adapter**
Where deserialization begins	Reaches a polymorphic call such as hashCode(), toString() or compare()	The linked methods that form the gadget chain	Typically reflective invocation that delivers the payload's impact

A typical deserialization execution flow looks like this:

```
HashSet.readObject()                // entry point (source)
 -> HashMap.put() -> HashMap.hash()
  -> TiedMapEntry.hashCode()        // trampoline: Object.hashCode()
   -> LazyMap.get()
    -> ChainedTransformer.transform()
     -> InvokerTransformer.transform()
      -> Method.invoke(Runtime.exec)  // sink
```

16 of ysoserial's 34 payloads (47%) are reflection-based, and all of them depend on the JDK-internal TemplatesImpl sink. Strong encapsulation of JDK internals (JDK 16–17) made TemplatesImpl unreachable, so all these gadget chains fail on modern JDKs unless the target runs with --add-opens.

The research community has since moved to successors such as GadgetBuilder, which includes around 60 main gadget chains with several variations and combinations of more than 300 payloads, and to automated chain discovery, covered later in this paper.

Some chains contain no third-party gadgets at all; they rely only on JDK classes. Just the presence of a vulnerable JDK version and a deserialization endpoint is enough for an attacker to achieve RCE. Historically these are called **golden gadget chains**. Public examples include Jdk7u21 (JDK 7u21 and earlier) and URLDNS, a JDK-only chain that triggers a DNS lookup and is widely used to blind-detect deserialization endpoints. Although Oracle constantly patches the JDK and blocks or removes vulnerable classes, research has shown that **certain JDK classes can be reintroduced by libraries**. In 2026, Atredis Partners showed that applications shipping their own copy of Xalan outside the java.xml module bring the TemplatesImpl sink back on JDK 21.

The possible gadget chains available in an application are based on the cumulative collection of all loaded classes and all transitive dependencies of the application (Haken, Black Hat 2018).

The phases of a typical deserialization attack are as follows:

- A vulnerable application accepts user-supplied serialized objects, directly (HTTP body, cookie, session file, RMI/T3/IIOP/P4 port, JMS message) or indirectly (a JNDI lookup, a database query, or a nested SignedObject).
- An attacker creates a malicious gadget chain with a payload that triggers a security-sensitive call, typically unsafe reflection operations that achieve RCE, serializes it into a stream of bytes, and sends the crafted data to the application.
- The vulnerable application reads the received stream of bytes and tries to construct the object by deserializing it.
- During (or shortly after) deserialization, the gadget chain executes, and the system is compromised.

What is the impact of such a system compromise?

The impact of a deserialization attack depends on the creativity of the attacker. Depending on the payload, a gadget chain can perform remote code execution, remote command execution, arbitrary file write or delete, server-side request forgery, NTLM relay and denial of service. Unauthenticated deserialization RCE vulnerabilities are routinely rated critical. In practice, these bugs now go straight from advisory to ransomware or mass data-theft extortion, often within hours of disclosure.

Where exactly is the vulnerability, and what makes a system exploitable?

There are two criteria for introducing a deserialization vulnerability to an application:

- The application must accept and deserialize serialized data from a location to which an attacker has access, including direct entry points such as HTTP requests and indirect locations such as a session store, a message queue or a remote naming service.

- Gadget classes must exist on the classpath or module path of the application. The application does not need to use them; the classes only need to be loadable.

Thus, versions of a library that has vulnerable classes (gadgets) are not enough on their own to introduce the vulnerability. The application must also deserialize data from an unsafe location, which is the root cause of the vulnerability.

This is precisely why the Apache Software Foundation did not consider InvokerTransformer itself to be the vulnerability and did not assign a CVE to it.

The presence of a gadget class in the classpath alone does not introduce a deserialization or RCE vulnerability; it makes the deserialization vulnerability exploitable. **It is the combination of both criteria that makes the application vulnerable and exploitable.**

Nevertheless, Apache hardened the library in two ways:

- **Commons Collections 3.2.2** makes deserialization of eight unsafe functor classes, including InvokerTransformer, throw UnsupportedOperationException by default. Setting the system property org.apache.commons.collections.enableUnsafeSerialization=true restores the old behaviour.
- **Commons Collections 4.1** removed Serializable from those classes entirely, breaking backwards compatibility by removing any way to turn it back on.

With the 3.x fix, a system cannot both serialize these functors and be protected at the same time. Developers must configure a JVM-wide system property and either risk breaking functionality or remove the deserialization gadget. With serialization removed from InvokerTransformer, attackers can no longer use it to craft malicious gadget chains.

03 Does this Solve the Problem at its Core?

As the saying goes, treating the symptom rather than the cause is no solution, much like what happened with InvokerTransformer. Disabling its serializability is not the proper way to solve the problem. It can break applications that legitimately serialize the class, and it does not make an application that deserializes untrusted data secure. InvokerTransformer is not the only known gadget. Researchers have identified many others and will keep identifying more, especially now with the help of AI.

04 Variations of Deserialization Attacks

Deserialization attacks come in several variations, each with its own execution mode and impact. Four of them are classic and the last decade added three more.

- **Blind deserialization attacks** target systems behind a network firewall that blocks outgoing connections. Attackers use JDK-only probes such as URLDNS to confirm that an endpoint deserializes untrusted data and then use timing, error or file-write side channels to exfiltrate data or even perform chained SSRF attacks.
- **Asynchronous (or stored) deserialization attacks** store the gadget chain in a database, message queue, cache or file, and it executes when the target later reads and deserializes it.

Apache Tomcat CVE-2025-24813 is a textbook case: attackers used a partial PUT to write a gadget chain into Tomcat's file-based session store, then submitted GET requests with the matching JSESSIONID that triggered the deserialization.

- **Deferred-execution deserialization attacks** do not execute the gadget chain during deserialization but after it completes. Historically this used finalize().

  Finalization was deprecated for removal by JEP 421 in JDK 18 but is still enabled by default, and can only be turned off with --finalization=disabled.

  Modern deferred chains also fire when the application later calls toString(), hashCode(), equals() or a proxied interface method on the returned object, for example when logging it.
- **Denial-of-service deserialization attacks.** Vendors long dismissed deserialization DoS as "will not fix"; JEP 290 changed that by adding maxdepth, maxrefs, maxbytes and maxarray limits to the serialization filter. These limits are not enabled by default. Unless someone explicitly sets jdk.serialFilter with limits or installs a filter, a JDK-only nested-collection payload can still pin a CPU or exhaust memory. A mitigation that exists only in configuration is a mitigation many production systems do not have.
- **Nested (re-entrant) deserialization attacks** hide the real payload inside an allowed wrapper that deserializes again. java.security.SignedObject.getObject() opens a fresh ObjectInputStream over its inner bytes; a custom allowlisting ObjectInputStream subclass on the outer stream does not apply to it. This exact pattern turned a hardened GoAnywhere MFT code path back into CVE-2025-10035.
- **Lookup-delivered deserialization attacks** reach readObject() through a naming or directory lookup instead of a request body. After JDKs stopped trusting remote codebases (RMI since 8u121, LDAP since 8u191), Log4Shell (CVE-2021-44228) payloads switched to returning a serialized gadget chain in the LDAP javaSerializedData attribute, which the client deserializes locally.
- **Supply-chain (dormant) gadget chains** are incomplete chains already sitting in popular libraries, one small code change away from becoming exploitable. Sleeping Giants (ACM CCS 2025) shows three such changes: making an interface or superclass serializable, which silently turns every subclass into a potential gadget; moving a constant into a final field, which deserialization can overwrite; and adding a serializable class that calls an innocuous JDK interface method such as Runnable.run() or Iterator.hasNext(), which can live in a completely different dependency. The change can be accidental or a deliberate supply-chain attack. Either way, the next payload that reaches an insecure readObject() finds a complete chain waiting.

05 How Can Security Teams Protect Against Deserialization Attacks?

How can production systems be protected if vendors cannot ship patches fast enough, DevOps teams cannot keep up with the patching cadence, and development teams cannot change the source code of the whole stack, including third-party libraries and legacy components?

Web Application Firewalls. WAFs have no application context and only see the application's input and output. They typically look for the Java serialization magic bytes (AC ED 00 05, or rO0AB in Base64) in HTTP requests. In practice, payloads arrive encoded, compressed, encrypted or executed in two steps. The Tomcat CVE-2025-24813 payload was Base64-encoded and uploaded in one request, then triggered by a second request. In GoAnywhere CVE-2025-10035, the gadget chain travels inside an encrypted response, so a WAF sees nothing but opaque bytes.

A JNDI-delivered chain never crosses the WAF at all: it arrives as the response to the server's own outbound LDAP lookup, served from the attacker's directory.

> **Any solution to deserialization that has no application context or operates outside the application cannot adequately solve the deserialization vulnerability.**

Java agents that disable deserialization completely, or runtime security (e.g. RASP) products that apply custom blocklists or allowlists to classes. Completely disabling deserialization system-wide breaks every application that relies on it for common functionality such as session replication, caching, RMI/JMX or messaging. Most enterprise apps would break, so this is not a viable solution for most cases.

Blocklisting depends on listing dangerous classes. Blocklists require profiling to confirm the application does not use those classes legitimately; otherwise the risk of breaking functionality is significant. Applying a negative security model means we can never be sure everything is blocked. The list has to be maintained constantly and, by definition, does not protect against unpublished, zero-day or AI-generated chains. Any security solution that promotes blocklisting as the answer is playing a never-ending game of Whack-a-Mole, and the last few years have made that game measurably worse for defenders:

- **All blocklists are out of date by default.** GadgetBuilder (NordSec 2025) combines 60 main chains with interchangeable trampolines and sink adapters into 303 payloads. AI security tools have made gadget finding easier than ever. The catalogue of known gadgets keeps growing and will never stop.
- **Real-world blocklists decay.** GadgetBuilder tested seven real-world deserialization filters (SerialKiller, NotSoSerial, MogwaiLabs and the lists shipped in Apache Ignite, Kafka, Fury and Dubbo) against 303 payload variants. Every list blocked a smaller share of GadgetBuilder's payloads than of ysoserial's. Ignite's list stopped only 21% of the variants and let 54 RCE payloads through; Kafka's SafeObjectInputStream let 34 through. Even Dubbo's blocklist, which blocked every ysoserial chain, let 15 non-RCE payloads pass. **Blocklists of known payloads are not future-proof.** They were built to stop ysoserial's chains, and ysoserial stopped adding chains in 2021. Any filter configured against a frozen catalogue gives a false sense of completeness the moment someone finds a new gadget or swaps a trampoline or sink.
- **Java platform hardening is routed around.** Although JDK changes had broken 20 ysoserial gadget chains on modern JDKs, by swapping trampolines and sinks GadgetBuilder revived 17 of them (85%) on OpenJDK 16 to 24, without --add-opens on the target.

To make things worse, JDK-only chains cannot be blindly blocked or removed, because the application will most probably break without the required functionality. And transitive dependencies

create library sprawl, including shaded and relocated copies of the same classes under new package names, which makes identifying and disabling "dangerous" classes even harder.

Allowlisting is a much better approach, and JEP 290/415 now make it a first-class JDK feature. But building the allowlist again requires profiling, and the result is an extensive list of classes that must be re-derived for every release and every dependency upgrade. That friction usually leads to stale allowlists that cause false positives, or to over-broad wildcards that let chains through.

Even a perfect allowlist cannot stop a chain assembled from allowed classes, cannot follow nested deserialization through a wrapper class it lets through (e.g. the GoAnywhere SignedObject case), and does not stop JDK-only denial of service unless resource limits are also configured.

Also, **allowed classes or dependencies can introduce gadgets after an upgrade.** A dependency that is gadget-free today can become one tomorrow. The Sleeping Giants study (ACM CCS 2025) found that whether a class is serializable fluctuates strongly as a library evolves, and that tiny, innocent-looking edits, such as adding Serializable to a class, are enough to form a complete gadget chain. When the researchers applied such edits to 533 Maven dependencies, three state-of-the-art detectors reported new gadget chains in about 26% of them, and manual verification confirmed dormant chains in 53, ready to be activated by an accidental change or a deliberate supply-chain attack. The practical lessons:

- A clean dependency scan is a point-in-time result, and any routine minor-version bump can quietly reopen the door.
- Allowlists of classes should be re-reviewed after any of these classes are refactored or upgraded.

Blindly blocking process execution and file/network I/O. This reduces the impact of an RCE chain, but it does not stop denial of service, does not stop chains whose sink is SSRF or NTLM relay through URL.openStream(), and, applied blindly to the whole process, blocks legitimate application behaviour as well.

OS-level controls such as seccomp, AppArmor and container profiles remain valuable defence in depth, but they see system calls and file paths, not Java call stacks. They can block a capability the application never uses, such as process execution, but they cannot tell the application's legitimate use of a capability from a gadget chain abusing it, and they are blind to attacks that never leave the JVM, such as malicious bytecode loaded into memory.

06 So, What is the Proper Fix?

Is there a solution that completely solves the problem, stops all the various types of deserialization attacks, avoids the Whack-a-Mole game and works out of the box without depending on configuration?

According to CERT, the proper solution to native Java deserialization requires "developers to re-architect their applications" and remove native Java deserialization altogether. Such a fix requires significant code changes, time, effort and money. If changing the source code and the application's architecture is an option, it remains the preferred approach: replace native serialization with a

data-only format bound to explicit types, and prefer Java records, whose deserialization always goes through the canonical constructor.

However, even if an application does not perform any deserialization in its own components, most servers, frameworks and third-party components do: session persistence, clustering, caches, RMI/JMX, message brokers, license handlers. It is practically impossible to confirm with 100% certainty that the whole stack does not and will never perform native Java deserialization without breaking existing required functionality.

Enterprises need protection fast, without source code changes, complex configurations or fine tuning. For organizations running hundreds of instances, many of them commercial products and dependencies they cannot modify, a code change per vulnerability is rarely feasible inside the exploitation window. That window is now measured in hours. CVE-2025-24813 in Tomcat was exploited in the wild 30 hours after disclosure.

CERT suggests that blocking the network port using a firewall might sometimes solve the problem. In most cases it is not applicable or practical, because the deserialization endpoint shares the application's main port. Tomcat CVE-2025-24813 and GoAnywhere CVE-2025-10035 were both reached over the normal HTTP(S) listener, and blocking it renders the server useless. Port restriction does still help for dedicated protocol ports such as WebLogic T3/IIOP, SAP P4 or ActiveMQ OpenWire (61616), and should be applied there, depending on the environment.

How did the Vendors of the Affected Systems Solve the Issue?

Most vendors added deserialization filters. However, the CVEs kept coming despite the filters. Following the ysoserial release in 2015, almost every affected vendor implemented the same security controls: they removed native deserialization, added blocklists of dangerous classes, or adopted allowlists such as JEP 290 filters.

Despite all this effort, recent CVEs showed that in the very same, patched, products, attackers kept finding a way through. Below are six cases where filters, allowlists or blocklists were in place, but researchers kept finding new deserialization CVEs anyway.

PRODUCT	CONTROL IN PLACE	WHAT FOLLOWED	WHY THE CONTROL FAILED
Apache Camel	Apr 2026: JMS ObjectMessage class filter added after CVE-2026-40860 (CVSS 9.8).	Jul 2026: CVE-2026-43866, a bypass of that filter roughly ten weeks later.	The vulnerable code path applied no filter at all. The fix then trusted the whole org.apache.camel.** namespace, so a forged Camel DefaultExchangeHolder passed the allowlist and injected message body, headers and properties using only java.lang/java.util types. No gadget chain needed.
Fortra GoAnywhere MFT	2023: type allowlist on ValidatingObjectInputStream after CVE-2023-0669 (Cl0p mass exploitation).	Sep 2025: CVE-2025-10035 (CVSS 10.0). Exploited by Storm-1175 for Medusa ransomware.	The allowlist admitted java.security.SignedObject, whose getObject() opens a second, unfiltered ObjectInputStream (watchTowr).

PRODUCT	CONTROL IN PLACE	WHAT FOLLOWED	WHY THE CONTROL FAILED
Oracle WebLogic	Class blocklists and filters on T3/IIOP since 2015.	Since 2015 Oracle has shipped dozens of fixes for WebLogic deserialization bugs that researchers keep bypassing. It started with CVE-2015-4852, which was bypassed in CVE-2016-0638 and then again in CVE-2016-3510. Read more about these bypasses in Waratek's blog post.	Oracle's fixes relied on a class blocklist in the T3 handler, which only rejects classes from known exploits. Researchers wrapped the same gadget chain in StreamMessageImpl or MarshalledObject, which deserialize their contents through a second, unfiltered stream. JRMP callbacks fetched the payload over an outbound connection, and when Oracle blocked the Registry interface, attackers used the Activator. When old classes were blocked by the filter, researchers switched to Coherence gadgets. The XMLDecoder fix, which blocked specific XML elements, failed the same way.
Oracle WebLogic	Class blocklists and filters on T3/IIOP since 2015.	More recent examples of WebLogic deserialization bugs that researchers keep bypassing. 2023: CVE-2023-21839. 2024: CVE-2024-20931, a bypass of CVE-2023-21839. 2025: CVE-2025-21535 (CVSS 9.8).	In CVE-2023-21839, attackers bind a serialized ForeignOpaqueReference with malicious field values, and its getReferent() performs an attacker-controlled JNDI lookup. The original fix validated only lookup() arguments and left the deserialized JNDI environment unvalidated. CVE-2024-20931 bypassed it through InitialContext factory initialization, and CVE-2024-21006 through object factory resolution. WebLogic's class filters couldn't mitigate these because every class involved is legitimate.
Apache Dubbo	Serialization blocklist and serialization checks.	2023: CVE-2023-23638 (gadgets bypass), CVE-2023-29234 (serialization checks bypass), CVE-2023-46279 (blocklist bypass). All CVSS 9.8.	Three bypasses of the same control in one year. CVE-2023-29234 was reported by the JDD team, whose tool automates exactly this kind of gadget search. In 2025, GadgetBuilder still produced 15 payloads that passed Dubbo's filters.
XStream	Blocklist until 2021, then allowlist by default from 1.4.18.	2022: CVE-2021-43859 and CVE-2022-41966. Nov 2024: CVE-2024-47072 (DoS).	Dozens of 2021 RCE CVEs forced the switch; the maintainers concluded a blocklist cannot be secured for general purpose. The allowlist then could not stop stack-overflow and recursive-collection DoS attacks, because the chains used only types any reasonable allowlist must permit, such as HashMap, HashSet and LinkedHashSet.
Spring and LangChain4j (AI stacks)	Jackson polymorphic-typing validators.	Aug 2026: CVE-2026-59285 (Spring for GraphQL). Sep 2026: CVE-2026-97869 (LangChain4j agentic).	In LangChain4j, Jackson default typing let the persisted input choose which class to instantiate (OSV). In Spring for GraphQL, exploitation depends on Jackson 2.x and abusable classes on the classpath. JVM serialization filters never see JSON at all (Sonatype).

Why do the serialization filters keep failing?

These are not careless vendors; they include some of the best-resourced engineering teams in the industry. The failures share root causes that come from the model, not the people:

- **Insecure by default.** Filters are opt-in. The JDK applies no jdk.serialFilter unless configured.
- **Trust is based on type or package name, not behaviour.** An allowlist like org.apache.camel.** or a broad "trusted" namespace allows every class in it, including ones that can be abused without any known gadget. The same mistake appeared outside Java in 2026: LangChain's Python load() was fixed for "overly broad" allowlists (CVE-2026-44843).
- **Stream-scoped filters guard one object graph, but not any nested serialized object.** A filter attached to a stream inspects every class in that stream's object graph, but not serialized payloads embedded inside it as raw bytes. Wrappers such as java.security.SignedObject carry a second serialized object and deserialize it later on a fresh ObjectInputStream the filter never sees. Jenkins was exploited this way in 2017 (CVE-2017-1000353), and the same pattern reached production again in 2025 in Fortra GoAnywhere MFT (CVE-2025-10035).
- **Every patch fixes one code path.** WebLogic, Dubbo and Camel all needed a fix for the fix. Each patch mitigated the reported chain or code path, but the vulnerability remained reachable via other code paths.
- **Lists may still be vulnerable to zero-days.** Allowlists and blocklists are configured around expected application behaviour and known exploits, so they remain exposed to zero-days and newly discovered gadgets and trampolines.

  WebLogic's T3 blocklist had been extended repeatedly since 2015, yet CVE-2020-2555 used a new gadget chain in Oracle Coherence, which ships with WebLogic by default.
- **Allowlists must be reassessed every time the code changes.** Every release, dependency bump or library change alters which classes are reachable. Sleeping Giants showed dormant gadget chains activated by small code changes in 53 dependencies.
- **Teams who configure the filters do not always own the code.** JEP 415's own authors note that per-stream filters do not scale and cannot be imposed on deserialization inside third-party libraries. Enterprises running WebLogic, SAP or GoAnywhere cannot write those filters at all.
- **Attackers now have AI tools on their side.** AI tools collapse the slow parts of deserialization exploitation (finding endpoints, probing the classpath and discovering a gadget chain) into a fraction of the time they used to take.

The lesson of a decade of deserialization mitigations is not that someone wrote a bad list.

> **The lesson is that lists are incomplete by design. Lists are a necessary first line of defense, but when the raw material is potentially every class on the classpath, they cannot be the last nor the only defense.**

How the JDK Changed the Battlefield

Since 2017 the JDK team has added more deserialization-relevant controls than in the previous twenty years. Some materially help defenders. However, none of these controls has fully remediated the deserialization vulnerability class.

JDK RELEASE	CHANGE	EFFECT ON DESERIALIZATION DEFENSE
24 (Mar 2025)	JEP 486: the Security Manager was permanently disabled.	Disabling the Security Manager was the right thing to do, but there is no JVM sandboxing any more: runtime protection must come from an external agent.
24, 17.0.13, 11.0.25, 8u432	com.sun.jndi.ldap.object.trustSerialData defaults to false. LDAP javaSerializedData deserialization permanently disabled in 24 (JDK-8347714).	Closed the Log4Shell-era LDAP delivery path on patched JDKs; older JDKs remain exposed.
20; Oracle 17.0.17, 11.0.29	Protocol-specific jdk.jndi.ldap.object.factoriesFilter and jdk.jndi.rmi.object.factoriesFilter (JDK-8296666, JDK-8352614).	Restricts JNDI object-factory abuse by default.
18 (Mar 2022)	JEP 421: finalization deprecated for removal.	The classic finalize() deferred trigger can be disabled only if --finalization=disabled is set. Finalization remains enabled by default in current Java SE releases.
17 (Sep 2021)	JEP 415: context-specific filters via a JVM-wide filter factory.	Filters can now vary per call site and library, but are still class lists written and maintained by the application owner.
17 (Sep 2021)	JEP 403: strong encapsulation of JDK internals; --illegal-access removed.	Blocked the default TemplatesImpl sink, but GadgetBuilder revived 17 chains using other sinks, and shaded Xalan restored TemplatesImpl outside java.xml.
15 (Sep 2020)	BadAttributeValueExpException.readObject() no longer calls toString().	Five ysoserial chains broke on JDK 15+. In 2024, the JDD gadget-chain detector found three new paths to the toString() trampoline.
9, 8u121, 7u131, 6u141	JEP 290: ObjectInputFilter, jdk.serialFilter, graph limits.	JVM-wide deserialization allowlists and DoS limits, but nothing is filtered unless configured.

JEP 415 deserves a closer look because it is the JDK's best answer to date. A filter factory is invoked for every new ObjectInputStream and can merge a per-library or per-thread filter with the JVM-wide one, using ObjectInputFilter.merge() and rejectUndecidedClass() to reject anything not explicitly allowed. Even the JEP's own discussion drew the obvious critique: it adds complexity to "the game of de-serialization whack-a-mole" (jdk-dev, May 2021).

Three architectural limits remain for these JVM filters:

- **They only control ObjectInputStream.** The JVM filters do not apply to XStream, Hessian, Kryo, SnakeYAML, polymorphic JSON and custom wire formats such as ActiveMQ OpenWire.
- **They make decisions based on class names, not deserialization behaviour.** An allowlist can only contain classes the application legitimately needs, and those classes can themselves be chained. GadgetBuilder shows real-world filter lists degrading as new gadgets appear.
- **Defining a deserialization list is a commitment.** Someone has to write, own and maintain it until the end of the app's lifecycle, enumerating every class the whole stack legitimately deserializes, for every release, across every version.

07 How AI Changed the Game

AI has not changed how deserialization vulnerabilities manifest or are exploited, but it is removing the scarcity of gadget chains that made CWE-502 feel manageable. Finding a new gadget chain used to be slow, specialist work by expert researchers. In 2026, GadgetHunter (FSE 2026), a neuro-symbolic research tool, reported a cost of about $0.04 and 23.5 seconds per candidate chain, and an AI coding agent confirmed 17 chains against hardened application servers in two days. At the same time, Mandiant's M-Trends 2026 estimates the mean time to exploit at negative seven days.

> **Effectively, we're now in the era where exploitation can start before a patch exists.**

The table below shows the distinct ways AI is now applied to deserialization research, each targeting a step that used to need a skilled human. Together, they show AI accelerating every stage of the attack pipeline, from finding a candidate chain to proving it exploitable.

CATEGORY	TOOL OR STUDY	TECHNIQUE	KEY RESULT
LLM-assisted gadget chain discovery	GadgetHunter (ACM FSE)	Static taint analysis finds candidate chains, the LLM judges whether each dynamic-dispatch hop is semantically reachable, and an SMT solver checks feasibility.	197 previously unknown chains validated as theoretically exploitable. 12 extra variants for Tomcat CVE-2025-24813. $95 for 2,666 candidates.
LLM-assisted taint analysis	Atredis Partners, *Finding Gadgets Like It's 2026*	An LLM coding agent (Claude Code) built payloads, ran them, read the failures and iterated, so discovery and exploit confirmation happened together.	17 confirmed chains in 2 days, 6 believed novel, including a JPMS bypass on JDK 21 via shaded Xalan in WildFly.
LLM-assisted fuzzing	LLM-JDFuzz	The LLM generates the serialized payload and repairs it over repeated attempts, targeting the hard step of turning a known chain into a valid object graph.	70.6% payload success on 34 ysoserial chains; 66.7% on chains newer than the model's training cutoff.
Autonomous AI exploit generation	Exploit-intel, CVEForge	An autonomous pipeline reverse-engineered an undocumented RPC protocol, mapped the Hessian wire format byte by byte, and searched for a gadget chain after a class blacklist blocked all known gadgets.	Bypassed the blacklist in Apache HugeGraph (CVE-2025-26866) with a JDK-only gadget chain in four hours for just $49.

What this means for defenders

- **The gadget catalogue is effectively unbounded.** Every published chain can have cheap variants. GadgetHunter found 12 additional exploitable variants for the single chain documented in the Tomcat CVE-2025-24813 advisory.
- **Blocklists and allowlists fall further behind.** Any security control that enumerates known classes is now racing against automated gadget chain generators.
- **Patch-first strategies are strategically late.** With a negative mean time to exploit, there is usually no patch to apply when the first attack arrives.

- **Defenders can use the same tools,** and should run them in CI. But finding chains in your own classpath produces more entries for a list; it does not stop the chain an attacker finds tomorrow. The durable control constrains what deserialization is allowed to do, not which bytes it receives.

08 A New Solution: Declarative Security-as-Code Deserial Micro-Compartmentalization

Based on the above, there's a clear lesson:

We cannot win the war against deserialization by simply filtering dangerous classes. We can only win the war by controlling what deserialization is allowed to do.

How Dynamic Micro-Compartmentalization works

Waratek treats deserialization abuse as a privilege escalation (CWE-250) and API abuse (CWE-227) problem.

This is in line with CERT and MITRE guidance to minimize privileges before deserializing and to avoid dangerous operations during deserialization. Converting bytes into objects never legitimately requires escalated privileges or privileged operations such as payload-controlled Runtime.exec(), arbitrary file writes, outbound connections or unbounded CPU and memory.

The dynamic micro-compartment enforces exactly that. Whenever any component of the application, a third-party library, the server or the JDK itself begins deserializing data, the agent wraps the operation in a dynamic, temporary and restricted compartment on the executing thread. Inside it, deserialization can do everything it legitimately needs to: construct objects, run custom readObject() logic, and use classes such as InvokerTransformer that blocklists would otherwise block. The micro-compartment does not allow payload-controlled privileged platform functions such as command execution, file and network access, or code injection. Throughout the operation, it also meters memory, CPU and object-graph depth.

01 · ENTER	02 · ENFORCE	03 · EXIT
Compartment created	**Objects built**	**Privileges restored**
Any deserialization in the stack is wrapped in a temporary, restricted dynamic micro-compartment on the executing thread.	Objects are constructed in memory normally. In case of a privilege violation, a standard Java exception is raised and the malicious graph is discarded before any integrity compromise.	On completion the dynamic micro-compartment is removed.

If a gadget chain attempts a payload-controlled privileged action, or a denial-of-service payload exceeds its resource budget, the operation is stopped immediately. The caller receives a standard

Java exception, exactly as it would for any failed deserialization, so the application handles it gracefully. The malicious action never executes, and the malicious object graph is never handed back to the application. When deserialization completes normally, the micro-compartment is removed and the thread's privileges are restored, so protection applies only for the duration of each deserialization and nowhere else.

Because the boundary is tied to the deserialization API rather than to a port or protocol, the same protection applies whether the bytes came from an HTTP request, an RMI or T3 call, a JDBC call, a JNDI response, a session file, a message queue or a nested SignedObject. Because the protection is based on what the code tries to do, it needs no prior knowledge of the gadget chain. All types of gadget chains hit the same wall, including published, zero-day, AI-generated, golden, dormant, blind, deferred and stored gadget chains.

Waratek's Security-as-Code platform remediates Java deserialization attacks through the declarative Deserial rule. Turning it on protects the entire application stack, regardless of whether the deserialization vulnerability is located in the app, a library, a framework, a transitive dependency, a server or even the Java runtime itself. It protects against known and unknown (zero-day) deserialization attacks, without code changes, profiling, blocklists or allowlists.

The rule is expressed declaratively in Waratek's ARMR language:

```
app("Deserialization protection"):
  requires(version: ARMR/2.7)
  marshal("Block deserialization RCE"):
    deserialize(java)
    rce()
    protect(message: "Deserialization RCE blocked", severity: High)
  endmarshal
endapp
```

Following is a comprehensive list of requirements for the ideal solution to this critical vulnerability. If your mitigation does not satisfy all of them, it is not ideal against deserialization attacks.

All of the checklist items must be achieved without noticeable performance overhead. In other words, the solution must be production-ready. This list can be used by anyone evaluating a deserialization mitigation.

REQUIREMENTS CHECKLIST

Operations

- Must work with zero source code changes
- Must work with no application profiling
- Must work without class blocklists or allowlists, and without tuning in the common case
- Must allow applications to be updated or upgraded without re-profiling or redeployment effort
- Must work with existing hardware and the existing application stack
- Must allow applications to use "dangerous" classes and gadgets for legitimate functionality
- Must not break existing application functionality or binary compatibility
- Must make deterministic, behaviour-based decisions rather than heuristic or signature guesses, so verdicts are explainable and false positives and negatives are minimized

Platform

- Must not depend on the Security Manager, which no longer exists on JDK 24+
- Must run on modern JDKs with strong encapsulation (JDK 17+)
- Must support all the Java versions enterprises actually run, from legacy Java 6 to the current LTS

Protection

- Must protect against all known gadget chains (ysoserial, GadgetBuilder and successors)
- Must protect against unpublished, zero-day and AI-generated gadget chains with no configuration
- Must protect against JDK-only (golden) gadget chains
- Must protect against dormant chains activated by a dependency upgrade
- Must protect against gadget chains that vendors classify as "will not fix"
- Must protect against blind deserialization attacks
- Must protect against denial-of-service deserialization attacks
- Must protect both the `Serializable` and the `Externalizable` interfaces
- Must protect against nested deserialization (for example via `SignedObject.getObject()`)
- Must protect against attacks through any entry point: HTTP, RMI, T3/IIOP, P4, JMS, JNDI, session stores
- Must protect the entire application stack: the JDK, the server, the framework, the application and all dependencies
- Must protect against deferred deserialization attacks
- Must protect against stored and lateral deserialization attacks (databases, queues, session files)

09 Real-World Incidents and How Dynamic Micro-Compartmentalization Mitigates Them

The recent deserialization CVEs below were possible despite platform hardening, signatures and/or class filters already being present. In every native-deserialization case, the exploit had to make privileged calls using untrusted data from inside a deserialization boundary, which is precisely what dynamic micro-compartmentalization blocks.

VULNERABILITY	ENTRY POINT AND ATTACK PATH	WHY THE USUAL FIXES FAILED	HOW WARATEK MITIGATES IT
Log4Shell **CVE-2021-44228 · Dec 2021**	Log4j 2 evaluates ${jndi:...} lookups inside logged messages, so any attacker-controlled string that gets logged makes the application fetch and load or deserialize an object from the attacker's server, enabling RCE (NCSC).	Log4j is often buried as a transitive dependency, vendor product teams could not patch themselves, and interim mitigations proved incomplete. WAF signatures were bypassed with nested-lookup obfuscation, formatMsgNoLookups missed context lookups (CVE-2021-45046), and newer JDKs blocked remote class loading but still deserialized javaSerializedData gadget chains.	The JNDI client's deserialization of javaSerializedData is a boundary, so the gadget chain is stopped with no dependency on a Log4j patch and no knowledge of the chain.
Oracle WebLogic **CVE-2024-21216 & CVE-2025-21535 · Oct 2024 & Jan 2025 · CVSS 9.8**	Unauthenticated T3/IIOP RCE via crafted requests. NSFOCUS describes CVE-2024-21216 as insufficient filtering of incoming T3/IIOP data during deserialization (CIRCL).	A decade of lists and filter updates on the same protocols.	T3/IIOP object reconstruction runs through Java deserialization boundaries; gadget chains reaching privileged operations are terminated in the compartment, with no dependency on WebLogic's filter list being complete.
Apache Tomcat **CVE-2025-24813 · Mar 2025 · CVSS 8.6**	Exploited about 30 hours after the PoC. A partial PUT writes a Base64 ysoserial payload into the file-based session store; a GET with that JSESSIONID makes FileStore.load() call readObject().	A stored, two-step attack that WAF signatures typically fail to block because they lack context. Blocking the documented Commons Collections chain leaves the 12 extra variants GadgetHunter found.	Session loading triggers a deserialization boundary regardless of when the session file was written. The original chain and all its variants need privileged calls for their payloads, and each is stopped the same way.

The pattern across all CVEs is the same. The vendor fix arrived after exploitation began, the fix was specific to one entry point or known chains, and the next variant routed around it.

Waratek's security rule mitigates all deserialization vulnerabilities, including unknown zero-days, by asking one question: **"Is this deserialization operation trying to execute privileged operations with untrusted data that a deserialization operation never needs to do?"** Waratek's mitigation does not care which entry point, port, chain or variation is being used.

10 · CONCLUSION

Constrain Behaviour, Not Gadget Names

Nearly twelve years after ysoserial, Java unsafe deserialization is not a solved problem. Instead, AI made it a faster one. The JDK has given defenders real tools such as JVM-wide filters, context-specific filter factories, strong encapsulation and safer JNDI defaults. Attackers have answered with recombined chains, nested payloads, dormant dependencies and, since 2025, AI

systems capable of finding and weaponizing new chains for a few cents. Every control that works by naming known classes or known chains is now racing against an automated gadget chain generator, and the exploitation window has gone negative.

Waratek's deserialization security rule enforces a fundamental principle at runtime: **deserialization operations should be allowed to build objects in memory and nothing more.** Waratek's Security-as-Code platform enforces that principle at every deserialization boundary in the stack, without code changes, profiling or lists, so the next chain, human or AI discovered, has nowhere to go.

The attacker's advantage has always been the discovery of new gadget chains. AI just made discovery nearly free. So the only defense that scales is one that doesn't care what chain gets discovered. No lists of gadgets or classes to maintain, no configuration. Game over for all gadget chains and the unsafe deserialization problem.

Immutable Security

Say goodbye to regressions after deployments. Once your policy is defined, no code added to the codebase can supersede it, and a dependency upgrade that silently reintroduces a gadget chain still hits the immutable micro-compartment boundary.

Deployment Agnostic

Securing your runtime instead of your codebase or CI/CD pipeline lets critical protection be applied when the agent starts, rather than during the next deployment window. That matters when exploitation begins before the vendor patch exists.

Economically Scalable

Remove the toil of false positives, filter-list maintenance and long feedback loops between security and engineering, and transform the economics of AppSec at a time when AI has made the attacker's side cheaper than ever.

Ready to scale security with modern software development?

Take a guided tour to see how the declarative Deserial rule protects your Java estate, from Java 6 to JDK 25, against the gadget chains published today and the ones AI will find tomorrow.

Take the guided tour →

11 Selected References

Breen, "Finding Gadgets Like It's 2026," Atredis Partners
atredis.com/blog/2026/3/12/findings-gadgets-like-its-2026

GadgetHunter, ACM FSE 2026
kaixuanli-ecnu.github.io/papers/kaixuan-FSE26_GadgetHunter.pdf

Sleeping Giants, ACM CCS 2025
arxiv.org/abs/2504.20485

GadgetBuilder, NordSec 2025
abartel.net/static/p/nordsec2025-gadgetBuilder.html

Mandiant, M-Trends 2026 Executive Edition
services.google.com/fh/files/misc/m-trends-2026-executive-edition-en.pdf

Spring, "Spring and Security in the Times of AI"
spring.io/blog/2026/06/01/spring_and_security_in_the_times_of_ai

Sonatype, "91 Spring CVEs"
sonatype.com/blog/91-spring-cves-highlight-the-growing-ai-vulnerability-consumption-problem

LangChain4j advisory GHSA-gmwr-7wmf-mrjm
github.com/langchain4j/langchain4j/security/advisories/GHSA-gmwr-7wmf-mrjm

watchTowr, GoAnywhere CVE-2025-10035
labs.watchtowr.com/is-this-bad-this-feels-bad-goanywhere-cve-2025-10035

How Many Times Can You Patch the Same WebLogic Bug?, Manus Kennedy, Waratek 2026
https://waratek.com/blogs/weblogic-deserialization-patches/

The Hacker News, Apache Tomcat CVE-2025-24813
thehackernews.com/2025/03/apache-tomcat-vulnerability-comes-under.html

Mandiant, ShinyHunters Oracle PeopleSoft campaign
cloud.google.com/blog/topics/threat-intelligence/shinyhunters-renewed-mass-exploitation-campaign-targeting-oracle-peoplesoft

OpenJDK JEP 290 · JEP 403 · JEP 415 · JEP 421 · JEP 486
openjdk.org/jeps/415

Haken, “Automated Discovery of Deserialization Gadget Chains,” Black Hat 2018
i.blackhat.com/us-18/…/us-18-Haken-Automated-Discovery-of-Deserialization-Gadget-Chains.pdf

SEI CERT Oracle Coding Standard for Java, SER08-J
cmu-sei.github.io/secure-coding-standards/…/ser08-j

Apache Commons Collections security notes
commons.apache.org/proper/commons-collections/security.html

MITRE CWE Top 25 (2025)
cwe.mitre.org/top25/archive/2025/2025_cwe_top25.html