contact@namiq.io · namiq.io

NamiQ

SECURITY ADVISORY

Java4Shell

CVE-2026-114938 · OpenJDK HotSpot Base64

Assessment & Action Plan

A memory-corruption bug in OpenJDK's Base64 decoder reaches Jira, Keycloak and Jenkins, but only on x86-64 CPUs with AVX-512 VBMI. How to tell, and what to do.

PREPARED BY	Hung Le, NamiQ	DATE	19 November 2026
FACTS AS OF	19 November 2026	VERSION	1.0
SEVERITY	**High: CVSS 3.1 8.1 (Oracle); 9.2 under CVSS 4.0 (Calif)**	STATUS	Fixed 17 November 2026; not on CISA's exploited list; CISA: exploitation none

NamiQ © 2026 NamiQ. All rights reserved. This document is the property of NamiQ. It may be shared unaltered, with attribution.
Companion article: namiq.io/insights/java4shell-cve-2026-114938-openjdk-base64

On 17 November 2026 OpenJDK and Oracle shipped fixes for CVE-2026-114938, and the researchers who reported it, Calif, published their write-up under the name Java4Shell. It is a memory-corruption bug in the code HotSpot, the standard Java virtual machine, uses to decode Base64 fast. Java is meant to be the language where this class of bug does not happen. It happened underneath Java, in hand-written machine code.

Calif says it reached remote code execution, in default configurations, in Keycloak, Jira, Confluence, JFrog Artifactory, Jenkins and PingFederate without signing in, and in Apache Kafka, Apache Solr, Elasticsearch, Gerrit and Spinnaker after signing in. It also showed it in sample applications on Spring Boot and Apache Struts, and calls both lists non-exhaustive. Base64 is everywhere a server reads a login header, a token or an upload, which is why the list is long.

The bug is also narrower than that list makes it sound, and the narrowing fact is one you can check in ten seconds: it lives only in the code path for x86-64 processors with a specific instruction set, AVX-512 VBMI. Arm servers are not affected. This note is for whoever runs Java in production: what the bug is, which of your machines it can reach, what to do first, and what is still unknown. Everything below is as of 19 November 2026, and every claim links to its source at the end.

The short version

- Update every Java runtime to the release of 17 November 2026, then restart each JVM. A running JVM keeps the old code until it restarts.
- Only x86-64 CPUs with AVX-512 VBMI run the vulnerable code: Intel Xeon Ice Lake, Sapphire Rapids, Emerald Rapids and Granite Rapids, and AMD Zen 4 or newer (EPYC and Ryzen), per Calif. Arm64 is not affected. Check the CPU flags on each host, inside the VM, with the command in Step 2.
- Versions: OpenJDK's advisory marks the 17, 21, 25 and 27 release lines, and Calif says 17.0.6 and newer. The CVE record lists Oracle Java SE 8u511b33-perf, 17.0.21.0.3, 21.0.13.0.3, 25.0.5.0.3 and 27.0.1.0.3, Oracle GraalVM for JDK 17.0.21.0.3 and 21.0.13.0.3, and GraalVM Enterprise Edition 21.3.20.0.3 as affected. Calif names the builds made from OpenJDK as affected too: Corretto, Temurin, Microsoft, Zulu, Liberica, SapMachine, Red Hat, Oracle OpenJDK and GraalVM.
- If you cannot update today: start the JVM with -XX:-UseBASE64Intrinsics (Calif's mitigation). Only on x86-64: an Arm JVM refuses Calif's other flag, -XX:UseAVX=2, and will not start.
- Not known to be exploited as of 19 November 2026. CISA's exploited list (version 2026.11.18) does not hold it, and CISA's own assessment on 17 November was exploitation none, automatable no, technical impact total. That list records exploitation CISA knows of, so absence is not proof. Calif holds back its working chains and says it will publish them later.
- Our reading: plan for days, not months. Calif also says there is no universal exploit and every target needs its own chain, which is why this is a this-week job rather than a tonight job for most estates.

What the bug actually is

HotSpot compiles code that runs often into native machine code, and for some common operations it uses hand-written routines, called intrinsics, that use the processor's widest instructions. Base64 decoding has one for AVX-512. Calif found that it mishandles a malformed padded tail in the input: the output length it computes goes negative, and Calif says the routine miscalculates the store mask for the padded tail, so it writes a full 64-byte block, up to 62 bytes past the end of the result (Calif's figures). Java then rejects the input as invalid, but the neighbouring memory has already been overwritten.

The write is mostly fixed bytes, which is why it looked close to unexploitable. Calif's chain uses it to enlarge the length of a neighbouring array, turns that into a readable leak of the Java heap (Calif calls this stage JavaBleed: the heap can hold personal data, private keys, other secret keys and tokens) and then into writes that reach the JVM's compiled code and hooks, for code execution. The vulnerable path only runs after the JVM has compiled the decoder into native code, which happens on its own once a server has decoded enough valid Base64.

In plain terms: a few dozen bytes of memory corruption, on one kind of processor, in a routine almost every Java server calls with input a stranger sends. The first half of that sentence is why it is hard. The second half is why it matters.

The scores, kept apart: each source scores a different thing

Source	Score	What it says
Oracle (the CVE record)	CVSS 3.1: 8.1	Network, high complexity, no privileges, no user action; full confidentiality, integrity and availability impact. "Difficult to exploit"
Calif (the write-up)	CVSS 4.0: 9.2	Critical under the newer scoring system; no vector published
OpenJDK advisory	CVSS 3.1: 8.1	Highest-scored of the ten OpenJDK fixes that day; component hotspot/compiler
CISA (SSVC, 17 Nov 21:53 UTC)	Not a score	Exploitation: none. Automatable: no. Technical impact: total
FIRST EPSS (19 Nov)	0.0075, percentile 0.56	An estimate of the chance of exploitation in the next 30 days

The attack, stage by stage

From Calif's description of its general technique, without the parts it holds back. Calif names six stages; knowing them tells you where a defender can see something. The "what you can see" part of each step is our reading, not Calif's.

Our summary of Calif's six stages (warm, rehearse, trigger, disclose, repair, finish). The product-specific final step is not public.

Step	What happens
1. Warm	The attacker sends valid Base64 to a route that decodes it (a login header, a token, an upload) until the JVM compiles the decoder into the AVX-512 routine. No memory is corrupted. What you can see: nothing unusual.
2. Rehearse	An optional calibration round (Calif). What you can see: more of the same traffic.
3. Trigger	The attacker sends input with a malformed padded tail; the routine writes past its buffer and Java rejects the input. What you can see: rejected-Base64 errors in the application log.
4. Disclose	Heap contents come back in responses; Calif says the heap may hold personal data, private keys, other secret keys and tokens. What you can see: responses larger than the route normally returns.
5. Repair	The chain restores what it broke where it can; Calif sprays attempts and retries misses, and a wrong placement can crash the JVM. What you can see: hs_err_pid files and JVM restarts without a deploy.
6. Finish	An application-specific step turns the heap access into code execution or a chosen change. Calif withholds this per product. What you can see: it depends on the product; a JVM starting a shell is the clearest sign.

Calif says the remote attack is sensitive to network conditions: low, stable round-trip times are preferred, and it tested from a machine it rented in the same cloud region as its target. Our reading: the attack is not one request but a long series of them, and the failed tries leave crashes and rejected-input errors behind. Step 5 below is built on that.

Who is exposed, and who is not

Three conditions have to hold on the same machine. All three are things you can check, which makes this one of the easier critical Java flaws to triage.

Three conditions, all needed

Condition	How to tell	If it is not met
An affected Java runtime	java -version on the host, and inside each container image and vendor bundle	Not affected by this CVE
An x86-64 CPU with AVX-512 VBMI, visible to the JVM	All four flags avx512f, avx512bw, avx512vbmi and bmi2 in /proc/cpuinfo	The vulnerable routine is never used. Arm64 (Graviton, Ampere, Apple) is not affected
Untrusted Base64 reaching the decoder	Almost any web application: login headers, tokens, file uploads. Assume yes	Exposure is lower, but a client that connects to a hostile TLS server was also shown to be exploitable

Our reading of the CPU condition, not Calif's: the processor families it names are the ones behind most current general-purpose cloud instance types, so a fleet bought or launched in the last two or three years is likely to meet it. A VM sees only the flags its hypervisor passes through, so run the check inside the guest, not on a spec sheet.

Products that ship their own Java runtime are the easy ones to miss. Jira, Confluence, Artifactory, Jenkins images and many appliances carry a bundled runtime that your operating system's updates never touch. They are fixed by the vendor's own release, or by pointing them at a patched runtime where the vendor supports that.

Things people will get wrong

- Updating the package and not restarting. The JVM that is running keeps the compiled routine it already has. Until the process restarts, nothing changed.
- Trusting the spec sheet for the CPU. The flags a VM sees depend on the hypervisor and the instance type. Check inside the guest, per instance type.
- Setting -XX:UseAVX=2 everywhere. On Arm the JVM refuses it and does not start, so a fleet-wide setting takes down the Arm half of the fleet. -XX:-UseBASE64Intrinsics is the narrower flag.
- Editing a file and assuming the flag is on. Launchers, wrappers and container entry points each have their own place for JVM options. Read the running process with jcmd.
- Counting only the runtimes the package manager knows. Bundled runtimes under /opt and inside images are where the unpatched ones will be.
- Reading "not on CISA's list" as "safe". It means CISA has not recorded exploitation yet.

Step 1: find every Java runtime, including the bundled ones

```
java -version 2>&1
ls /usr/lib/jvm/ 2>/dev/null
# runtimes bundled inside applications (often under /opt, so no -xdev)
sudo find / -path /proc -prune -o -type f -path '*/bin/java' -print 2>/dev/null
# inside each running container
for c in $(docker ps --format '{{.Names}}'); do echo "== $c"; docker exec "$c" sh -c 'find /
-path /proc -prune -o -type f -path "*/bin/java" -print 2>/dev/null' || echo "   no shell in
$c: check this image by hand"; done
```

The host-wide search can take minutes, and can stall on a hung network mount. Then run each java it found with -version. A container with no shell (distroless and Jib-built Java images often have none) prints the "check by hand" line rather than nothing: an empty answer is not "no Java". On Kubernetes nodes without docker, use crictl or your image scanner for the same check.

Write down the full version string of each runtime. A distribution can ship the fix under a package version of its own: compare the installed package with your distribution's advisory, not only the java -version line.

Step 2: check the processor on each host

```
uname -m
grep -m1 '^flags' /proc/cpuinfo | tr ' ' '\n' | grep -E
'^(avx512f|avx512bw|avx512vbmi|bmi2)$'
```

This is Calif's check. All four flags must appear for the host to run the vulnerable code. aarch64 from uname means Arm: not affected. Do the check on each instance type you run, because a fleet often mixes generations.

Step 3: update, then restart every JVM

Install the 17 November 2026 update from whichever vendor supplies your runtime, then restart every process that uses it. For applications with a bundled runtime, install the vendor's release that carries the 17 November Java update; if the vendor has not shipped one yet, ask them for the date and apply Step 4 meanwhile. Rebuild container images from a patched base image and redeploy, then check the runtime in the running container, not in the build log.

Step 4: if you cannot update today

Turn off the vulnerable routine. Java then uses its ordinary Base64 code. Calif says the fast routine helps applications that decode enormous amounts of data; our reading is that most services will not notice the difference, but measure a service that decodes large payloads.

```
# add to the JVM options of each service on an x86-64 host, then restart it
-XX:-UseBASE64Intrinsics
# after the restart, confirm the running process has it (replace PID)
jcmd PID VM.flags | tr ' ' '\n' | grep -i base64   # expect -XX:-UseBASE64Intrinsics
```

Where the flag goes depends on how the service starts: its own JVM options setting (setenv.sh, a vendor's JVM options file, a systemd unit, a container's environment). The JAVA_TOOL_OPTIONS environment variable is read by the JVM itself, whichever launcher started it, so it is the catch-all when you cannot find the right file. Setting it in your own shell affects only programs started from that shell. Check the running process with jcmd, as above, rather than trusting the file. Run jcmd as the JVM's own user, and for a JVM in a container run it inside that container (docker exec CONTAINER jcmd 1 VM.flags); an attach error means the check did not run, not that the flag is missing. Calif's alternative, -XX:UseAVX=2, turns off all AVX-512 use and stops an Arm JVM from starting, so keep it off any mixed fleet's shared configuration.

What each stop-gap does and does not stop. Costs are our reading

Measure	What it stops	What it does not	Cost
Update and restart	The flaw	Nothing left of this CVE	A restart per JVM

-XX:-UseBASE64Intrinsics	The vulnerable routine is never used	A JVM where the flag did not reach the running process	Slower Base64 for very large inputs
-XX:UseAVX=2 (x86-64 only)	All AVX-512 code paths, including this one	Anything on Arm, where the JVM refuses to start	Broader slowdown than the narrow flag
Request buffering at a proxy	Not a general mitigation (Calif)	Calif: the Elasticsearch chain runs from one complete request	Little

Step 5: look for signs it was tried

Our suggestions, not Calif's, since the write-up gives no detection guidance. A failed attempt can crash the JVM, which leaves a file named hs_err_pid<number>.log in the working directory. A JVM that restarted without a deploy, or several such files in a few days, is worth a look. In application logs, a burst of rejected Base64 input (IllegalArgumentException from java.util.Base64) from one source is the other trace a trial run leaves. Calif's exploitation also needs many requests at a steady, low round-trip time, so the same source sending a long series of near-identical requests is a third.

```
sudo find / -path /proc -prune -o -name 'hs_err_pid*.log' -mtime -14 -print 2>/dev/null
```

If you find something: keep the crash files and logs before restarting or rebuilding anything, and treat what the JVM held in memory as read. That means rotating the session-signing keys, API tokens and passwords the service had loaded, not only patching it.

What to send your vendors today

> *We run [product, version] on x86-64 servers. Calif's Java4Shell write-up (CVE-2026-114938, 17 November 2026) names products that bundle an OpenJDK runtime. Which Java runtime does our version ship, is it affected, and which release of yours carries the 17 November 2026 Java update? Until then, do you support setting -XX:-UseBASE64Intrinsics in the product's JVM options, and where?*

Check every server and every project with lattice-scan

Steps 1 to 5 are done by hand, one server at a time. To do the same over a fleet, and to keep a record, use the open-source lattice-scan command line. Run on the server itself, it reads the installed packages, snaps, the hardware, the running container images with their operating system and packages, and the kernel, compares them with the CVE ids you name, and sends only those ids out.

Naming the CVE needs no key. It lists this CVE in our recent-CVE list through 23 November, so lattice-scan recent shows it too.

```
# on each Java server, after updating and restarting
lattice-scan --cve CVE-2026-114938
# or: everything recent, exploited or critical
lattice-scan recent
```

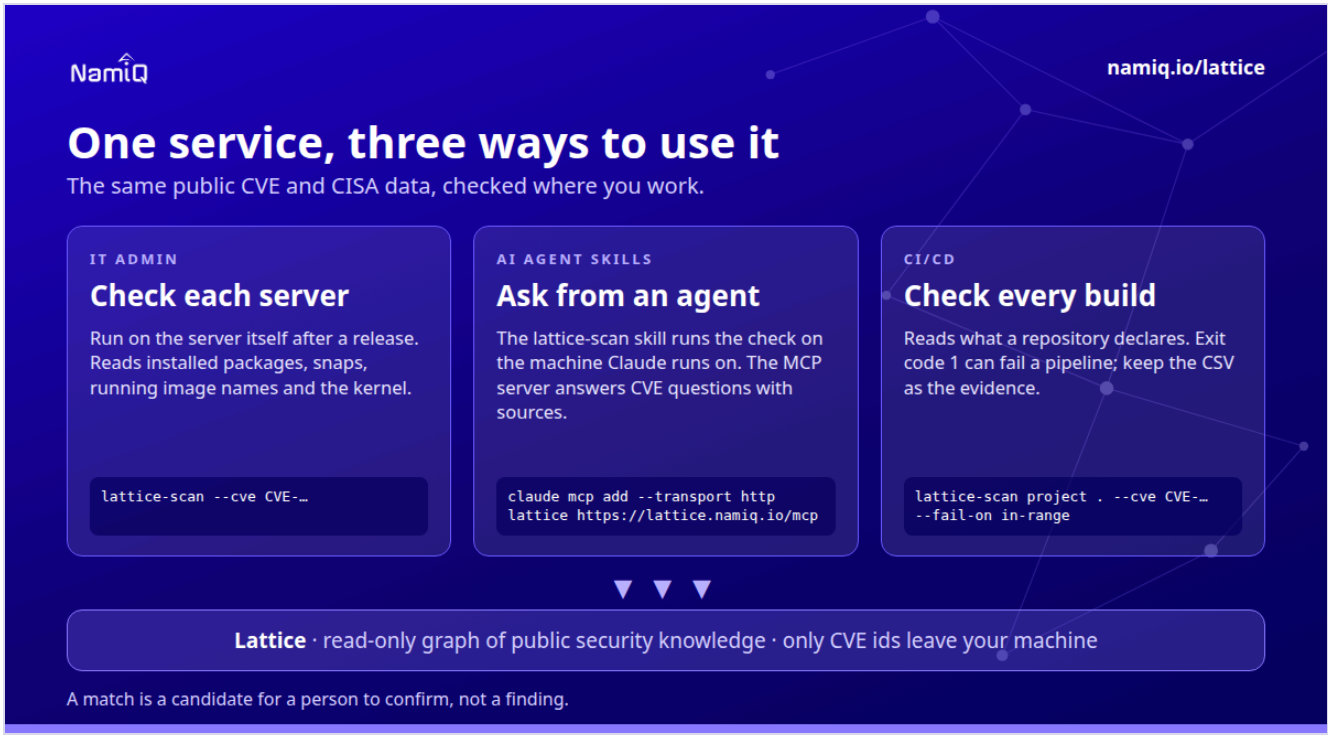
A match says an OpenJDK package is installed, not that the host is affected: the tool does not read CPU flags, so Step 2 is still yours, and a runtime bundled inside a product does not show. Naming the id gives only a name lead, not a version check, because the record lists Oracle's own builds rather than OpenJDK versions. By default it opens the images your containers are running; a deep scan (--deep) also reads the stopped images and the Java packages inside images. One thing that does not flatter us: when this note was first published on 19 November, our list left this CVE out, because it held only records scored 9.0 or more and Oracle scored this one 8.1. The list's bar is now 8.0.

For the code you ship, the project check reads what a repository declares, including Dockerfiles and compose files, such as a FROM eclipse-temurin:21 line. A tag without a patch number cannot show whether the fix is in, so confirm the image that is actually running.

```
lattice-scan project . --cve CVE-2026-114938
```

Who can use it, and how

For	What to do	Command
IT admin	Run the host check on each server after a release, and again when a new entry lands.	lattice-scan --cve CVE-2026-114938
AI agent skills	The lattice-scan skill runs the check on the machine Claude runs on. The Lattice MCP server answers questions about the CVE with its sources.	claude mcp add --transport http lattice https://lattice.namiq.io/mcp
CI/CD	Check the repository on every pull request. Exit code 1 fails the pipeline; keep the CSV as the evidence.	lattice-scan project . --cve CVE-2026-114938 --fail-on in-range



Where the same check is used. The three ways read the same public data; only CVE ids leave your machine.

What we do not know yet

- The exact fixed version for each vendor build. The record names the last affected Oracle builds of each line (27.0.1.0.3, 25.0.5.0.3, 21.0.13.0.3, 17.0.21.0.3); read your vendor's release notes for the build that carries the fix.
- Why Oracle lists a Java SE 8 build. OpenJDK's matrix does not mark its 8 or 11 lines for this CVE, and Calif says 17.0.6 and newer, but Oracle lists one Java SE 8 performance build (8u511b33-perf) as affected. If you run that build, treat it as affected.
- When Calif publishes its product chains, and whether others reproduce them first. Calif says it holds them "until everyone has had a chance to upgrade".
- Whether anyone is using it. CISA's assessment on 17 November was exploitation none. That can change without notice.

If you take one thing from this: the patch is easy and the restart is the step that gets skipped. The hard part is the runtimes nobody counted, inside products and images, and the flag check per instance type. If producing that list takes more than an afternoon, fix that too: the next runtime-level flaw will ask the same question.

Appendix 1. Severity decoded

CVE-2026-114938 (Oracle, the CVE record): score 8.1, `CVSS:3.1/AV:N/AC:H/PR:N/UI:N/S:U/C:H/I:H/A:H` Network, no login and no user action, but high attack complexity ("difficult to exploit", Oracle). Our reading: the processor condition, the compiled decoder and a heap layout the attacker does not choose all have to line up. Calif scores the same flaw 9.2 under CVSS 4.0; it did not publish the vector.

Metric	Value	What it means here
Attack vector	Network	Reachable over a network, with no need to be on the same segment.
Attack complexity	High	Needs conditions outside the attacker's control.
Privileges required	None	No account is needed.
User interaction	None	No one has to do anything.
Scope	Unchanged	The score keeps the damage inside the JVM; our reading: the heap it leaks can hold keys and tokens for other systems.
Confidentiality	High	Total loss of confidentiality.
Integrity	High	Total loss of integrity.
Availability	High	Total loss of availability.

Appendix 2. Timeline

Date	Event
22 October 2026	The CVE identifier is reserved by Oracle (CVE record, 04:13 UTC).
17 November 2026	OpenJDK publishes its advisory and Oracle its patch, at 1:00 pm Pacific time (Calif). Calif publishes its write-up the same day and names the flaw Java4Shell.
17 November 2026	The CVE record is published, 21:32 UTC, CVSS 3.1 8.1. CISA adds its SSVC entry at 21:53 UTC: exploitation none, automatable no, technical impact total.
18 November 2026	CISA's Known Exploited Vulnerabilities catalogue, version 2026.11.18, does not list it.
19 November 2026	FIRST's EPSS gives 0.0075 (percentile 0.56): an estimate of the chance of exploitation in the next 30 days, not a measurement.

Appendix 3. Exposure by situation

These priorities are our judgement from the attacker prerequisites in the sources. They are not vendor ratings.

Your situation	Priority	Timeframe
An internet-facing Java service (Keycloak, Jira, Confluence, Artifactory, Jenkins, PingFederate or your own) on x86-64 with all four CPU flags	Critical	Update and restart this week, starting today; until then set -XX:-UseBASE64Intrinsics and confirm it with jcmd
Same, reachable only from inside your network or a VPN	High	Update this week; one compromised machine on that network is enough
Java that only connects out (clients, batch jobs) to servers you do not control, on x86-64 with the flags	High	Update this week: Calif shows a client opening TLS to a hostile server can be compromised
A product with a bundled runtime and no vendor fix yet	High	Ask the vendor for a date (message below), set the JVM flag in the product's own JVM options meanwhile
x86-64 hosts where the four flags are not all present	Low	Update at the next maintenance window; record the flag check with the date, per instance type
Arm64 hosts (Graviton, Ampere, Apple)	None for this CVE	Update with the next release; never set -XX:UseAVX=2 here, the JVM will not start
Java 8 or 11 from OpenJDK builds	Low	OpenJDK does not mark these lines; Oracle lists one Java SE 8 performance build (8u511b33-perf). If you run that build, treat it as Critical

Appendix 4. Definition of done

- Every Java runtime has been listed, including runtimes bundled inside products and inside container images, with its full version string and the date.
- Each x86-64 host and instance type has its CPU flag check recorded, run inside the guest.
- Every runtime on a host that meets the conditions carries the 17 November 2026 update, read from the running process, not from the package database.
- Every JVM has been restarted since the update, or carries -XX:-UseBASE64Intrinsics confirmed with jcmd until it can be.
- Each vendor with a bundled runtime has answered with a fixed release or a date, and the answer is filed.
- Crash files (hs_err_pid*.log) and unplanned JVM restarts since 17 November have been looked at on the exposed services, and the result is written down.

- A date is set to re-read Calif's page and CISA's catalogue, because Calif says it will publish its product chains later.

Sources

- Calif: Java4Shell, CVE-2026-114938 (17 Nov 2026): https://calif.io/java4shell
- OpenJDK Vulnerability Advisory, 17 Nov 2026: https://openjdk.org/groups/vulnerability/advisories/2026-11-17
- Oracle Critical Patch Update, November 2026: https://www.oracle.com/security-alerts/cspunov2026.html
- CVE record CVE-2026-114938 (Oracle; CISA SSVC entry): https://www.cve.org/CVERecord?id=CVE-2026-114938
- CISA Known Exploited Vulnerabilities catalog (version 2026.11.18: not listed): https://www.cisa.gov/known-exploited-vulnerabilities-catalog
- FIRST EPSS (scores read 19 Nov 2026): https://www.first.org/epss/