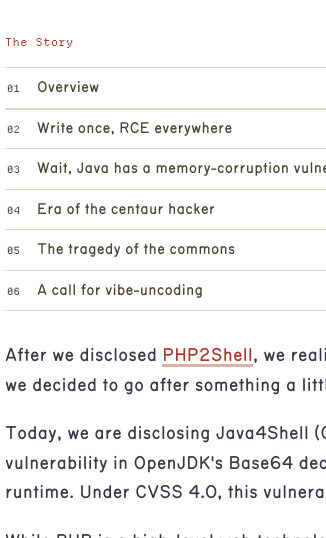


CVE-2026-114938

Java4Shell

Write once, RCE everywhere

Story by the New York Times

A memory-corruption vulnerability in OpenJDK's Base64 decoder. A remote attacker with no credentials can overwrite heap memory, read it in the style of Heartbleed, and reach remote code execution.

Begin ↓ Technical write-up ↘ Check your servers ↘ Recommendations ↘

The Story

01 Overview
02 Write once, RCE everywhere
03 Wait, Java has a memory-corruption vulnerability, in Base64?!
04 Era of the centaur hacker
05 The tragedy of the commons
06 A call for vibe-uncoding

After we disclosed PHP2Shell, we realized many people didn't know what PHP was. So we decided to go after something a little more familiar: Java.

Today, we are disclosing Java4Shell (CVE-2026-114938), a memory-corruption vulnerability in OpenJDK's Base64 decoder. OpenJDK is the most popular Java runtime. Under CVSS 4.0, this vulnerability scores 9.2 out of 10 (CRITICAL).

While PHP is a high-level web technology, Java sits much deeper. Java is the world's third most popular programming language, powering 90% of Fortune 500 companies. It is the beating heart of the modern economy. Banking, aviation, telecommunications, nearly every critical sector runs on Java, including national security systems.

Base64 is a very common way of representing binary data as text. It is used in authentication protocols, to encode file uploads, and to deliver network messages. To decode Base64 faster, OpenJDK uses specialized x86-64 CPU instructions to process many bytes at once. We discovered a bug in that optimized machine-level code that lets malformed input overwrite neighboring memory. With careful exploitation, the corruption can reveal sensitive memory, as Heartbleed did, or lead to remote code execution. The corruption itself can be triggered by just 64 `A` characters followed by a single `=`:

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=
```

In older OpenJDK 17, the trigger input is 128 of `A` followed by `=`.

Write once, RCE everywhere

Our initial testing demonstrated remote code execution in the following widely deployed Java applications, in their default configs.

Pre-auth RCE:

- Keycloak
- Atlassian Jira
- Atlassian Confluence
- JFrog Artifactory
- Jenkins
- PingFederate

Post-auth RCE:

- Apache Kafka Broker
- Apache Solr
- Elasticsearch
- Gerrit
- Spinnaker

The lists are non-exhaustive. We also demonstrated RCE in sample applications built on widely used frameworks, including Apache Struts and Spring Boot. Ordinary features such as authentication and file uploads can expose the vulnerable Base64 decoder, extending the potential impact far beyond the products we tested. samczsun also shared a client-side exploit with us. A Java program that opens a TLS connection to a malicious server can be compromised, even if it exposes no service of its own.

Here are a demo video of the exploits running:

Java4Shell RCE Demos: Keycloak, Jira, Confluence, Kafka, Elasticse
Calif
Watch on

Wait, Java has a memory-corruption vulnerability, in Base64?!

Yes. First time?

Memory corruption is the most common class of vulnerability on the Internet. But one of Java's biggest selling points is memory safety. Java manages memory on developers' behalf and checks memory accesses, protecting applications from the memory-corruption bugs common in other software.

But like PHP and many other technologies, much of Java itself is written in low-level languages like C, C++, and, in this case, assembly, where memory corruption is still a part of life.

To speed up Base64 decoding, OpenJDK uses specialized AVX-512 VBMI instructions on supported x86-64 processors. The bug sits in the hand-rolled assembly code that handles the final few characters. For the 65-character input above, OpenJDK allocates a 50-byte result array. After decoding the first 64 characters, the routine mishandles the stray = and calculates a negative output length. That invalid length becomes a full 64-byte write, sending 62 bytes past the end of the array. Java eventually rejects the input, but by then neighboring objects may already be corrupted.

Era of the centaur hacker

As with PHP2Shell, Calif used AI to find and weaponize Java4Shell. AI brings the horsepower. Our researchers bring the context. The best hacker today is neither AI nor human. It is the *centaur hacker*, part human, part AI.

It took our own Quang Luong an hour to discover the vulnerability, and two more hours to develop the first working exploit. The total token cost was less than $200.

One thing to note about this vulnerability is that there is no universal exploit. Depending on how Base64 is used and how the application works, different steps may be required for exploitation. Calif was able to accelerate the exploit development by spending initial effort building pipelines and strategies, then directing AI agents to use those for initial exploitation. Afterwards, most of the later chains are built almost autonomously with AI.

The tragedy of the commons

Java is one of many shared software stacks we all rely on. Over the years, it has grown to serve many different needs. Each feature request may make sense on its own, but everyone inherits the added complexity. Those who need a feature get the benefit. Everyone else carries the code and its risks, even if they never use it.

Java4Shell is a concrete example. The Base64 optimization helps applications that decode enormous amounts of data. But the code ships to everyone else too, including applications that gain little from it. On affected systems, an optimization few applications needed became a serious security risk.

Features can be useful, and most bugs can be fixed. But why do we have to worry about code we never needed in the first place?

A call for vibe-uncoding

If you don't need something, there is little reason to keep it running in your systems. Do you need that hand-rolled assembly Base64 decoder to process a few hundred bytes? Probably not.

Earlier this year we found Squidbleed, an older cousin of Heartbleed, hiding in the Squid proxy since 1997. The bug was in Squid's FTP module. But who uses FTP anymore? It's a dead protocol and should be removed.

Removing code is harder than it sounds. An apparently unused feature may have hidden dependencies. Someone has to understand what can be removed, make the change, and test what remains.

This is where AI can help to make that work practical at a scale we could not previously afford.

The same centaur hacker that discovered PHP2Shell and Java4Shell can comb a large codebase and flag dead code, bloated features, and needless optimizations. We call this *vibe-uncoding*: identify the parts a system actually needs, then remove everything else to reduce its attack surface.

With AI, everyone is asking, "What can we build?" But maybe the right question for security is, "What can we delete?"

The Write-up

01 The Tail of Base64
02 Discovery
03 The vulnerability
04 Exploitation strategy
05 Heap Feng Shui (Grooming)
06 Remote Heap Feng Shui
07 Warm-up, rehearsal, and retries
08 Limitations
09 PoC
10 Affected products
11 Recommendations
12 Credits

The Tail of Base64

By Quang Luong

On affected OpenJDK builds, malformed Base64 data can cause a constant-value overwrite of allocated heap memory. Through remote Heap Feng Shui, we turn that overwrite into a heap leak and remote code execution.

Discovery

A few months back, I was developing a 0-click exploit on a certain phone vendor. The bug was constrained: it is a linear out-of-bounds write and we have no way to bypass ASLR. I almost considered it unexploitable until I audited the Android Runtime. Turns out, it has some predictable memory allocation pattern and if you align the registers from the starting memory corruption just right you can make the runtime interpret attacker's bytecode.

> LESSON LEARNED
> Complicated system interactions are hard, and this complexity can make exploiting hard bugs easy.

After completing that exploit chain, inspired (and bored), I decided to look into some thing completely different. So I pulled the official JVM implementation (OpenJDK HotSpot) and started auditing. Java is a mature platform, so I was aware that having a "good" bug here would be hard, but I wanted to give it my best attempt anyway.

Recalling my earlier lesson that complex systems and code crossing multiple boundaries are difficult to get right, I focused the audit on the JIT compiler, intrinsics, widely accessible APIs, and cryptography. Well, more precisely, I told Codex to look at those areas like any serious security researcher in 2026. To be honest, I didn't expect to find anything trivially exploitable, perhaps only some logic flaws or side-channel issues.

Most early results Codex brought me were slops, either unreachable or the requirement of exploitation is not realistic at all. This looks like a gamble I am going to lose. But you know what, fortune favors the bold. So I just prompted harder...

Eventually, Codex brought me a finding for a memory corruption in the Base64 decoder itself. I came looking for copper and found gold, through sheer persistence. But wait, is it really gold, or just fool's gold?

> NOTE
> That sheer persistence is prompting AI and ignoring slops for around 20 minutes, by the way.

The vulnerability

The issue lies in OpenJDK HotSpot's x86-64 Base64 intrinsic. C2 is the most aggressive JIT compiler in the Java Virtual Machine. Specifically, if you trigger a function repeatedly, then it will compile that function into native code. Even then, ordinary native code itself is not fast enough. Modern CPUs have specialized instructions that allow processing data even faster. HotSpot uses them in optimized routines called intrinsics. The x86-64 Base64 intrinsic miscalculated the padded tail's store mask, causing an out-of-bounds write.

The x86-64 Base64 intrinsic iterates through the input in 256-byte blocks; if it is less than 256, then it will do so in 64-byte blocks. When the input is less than 64 bytes, it uses `evmovdqub`, the intrinsic for the `VMOVDQU8` instruction. This instruction takes a 64-bit mask that decides which bytes are read from or written to memory. The mask is first calculated from the input and output length, then updated to reflect the number of padding bytes (`'='`) in the Base64-encoded string. This is shown in the following (pseudocode):

```
COPY
// Zero High Bits Starting with Specified Bit Position
input_mask := bzhiq(-1, input_length)
output_len := (input_length / 4) * 3

// handle padding
if input[input_length - 1] == '=':
    output_len -= 1
    input_mask >>= 1
    if input[input_length - 2] == '=':
        output_len -= 1
        input_mask >>= 1

output_mask := bzhiq(-1, output_len)
```

In the above code, if the remaining `input` is of the form `=, ==, X=, X==, XY=`, or `===`, then `output_len` will be computed as zero. However, since there is padding, `output_len` is decremented, resulting in an underflowed value of `-1` or `-2`. Since `bzhiq(-1, output_len)` doesn't change any bits if `output_len >= 64`, this results in `output_mask == -1u64`.

Later the code uses `evmovdqub` to read and write multiple bytes at a time. For a write, if the mask bit is `1`, the byte at that index in the vector register is written to memory. So when `output_mask == -1u64`, all `64` bytes are written during the following operation:

```
COPY
evmovdqub(
    Address(dest, dp) /* Address: destination address addr */,
    k1 /* k1 == output_mask */,
    tmp /* XMMRegister src register/value to write */,
    true,
    Assembler::AVX_512bit /* vector length */
);
```

The generated stub returns the number of bytes it successfully decoded, and returns early when it detects invalid input. The remaining bytes are handled by the Java code, which throws `IllegalArgumentException` if the input is invalid.

At runtime, the decoded buffer length is calculated as `(ceil(inp_len / 4) * 3) - padding`. When the bug is triggered, with `65` bytes (`[64 bytes] + '='`) as input, that calculation allocates a `50`-byte buffer. During decoding the bug makes `output_mask == -1` for `evmovdqub`, so `64` bytes are written starting at offset `48`, an out-of-bounds write past the end of the buffer.

On OpenJDK 17, the trigger input is `'A' * 128 + '='` instead of `'A' * 64 + '='`, because the length threshold to reach the code path is higher.

Exploitation strategy

The Base64 bug was gold, but not the type that sits on the riverbed waiting to be panned out. It was the type that was deep, dark, and hard to access. While we had memory corruption, exploiting it looked difficult, perhaps even borderline unexploitable, since it only wrote fixed bytes to the heap. Only the landing position of the out-of-bounds write is controllable.

We get the following 64 bytes write starting from the position in buffer where the remaining sets of Base64 decoded bytes would have been written to:

- Full controlled write to byte 0 (in bound)
- 4-bit controlled write to byte 1 (in bound)
- Uncontrolled write to bytes [2, 63]. For bytes `48..63`, the value will always be `0x9a`.

The uncontrolled out-of-bounds write turned out not to be a big problem. We reused a common JavaScript engine exploitation primitive, the classic technique of overwriting a buffer length.

We call the buffer whose length we overwrite the `writer`. Once its length is corrupted, it keeps accepting our chosen bytes, so if we place another array or object after it, we can corrupt that neighbor with all of our chosen values. That upgrades the fixed write into a bounded, chosen-value linear write.

If we then corrupt another array's length and its data is copied back into the response, we have built a heap disclosure primitive, the `reader`, from the out-of-bounds read. At this point, before any code execution, we already had JavaBleed, the ability to leak contents from the Java heap, which may hold PII, private keys, other secret keys, and tokens.

From the `writer` and `reader`, the rest of this write-up builds the full attack. We repair the objects the linear write damaged to keep the JVM running, mount data-only attacks on application logic, and escalate the bounded write into arbitrary writes that reach the Code Cache and JVM hooks for code execution.

But first things first, we need to align the heap to our advantage.

> ALTERNATIVE STRATEGY
> Other alignments conceptually exist (e.g. type confusion, registers control from GC itself). Due to time constraints, we have not researched them deeply, and this strategy generalized well enough for our cases. Give us a shout if you invent something new.

Heap Feng Shui (Grooming)

G1 divides the heap into fixed-size regions. Before the region arithmetic matters, the critical arrays have to leave their threads' private allocation ranges.

HotSpot normally gives each Java thread a thread-local allocation buffer (TLAB), a private slice of Eden. Objects created by different request threads advance different cursors, which gives us no useful order between a Base64 decoder in one TLAB and a writer in another.

The first grooming layer controls the allocation route. The critical array's size and the space left in the worker's TLAB decide whether HotSpot refills that TLAB or takes the attacker-preferred outside-TLAB allocation path.

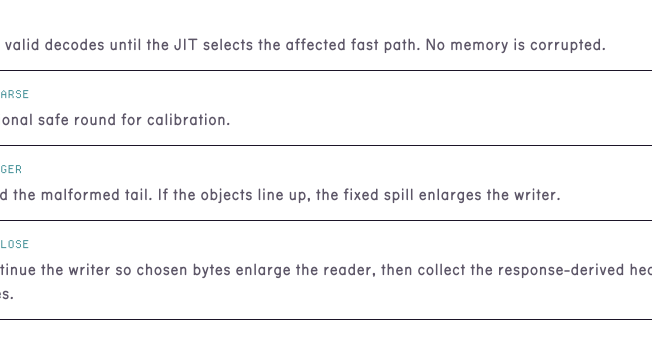
The second layer controls the layout. However, we do not choose how G1 allocates memory, and we don't know the current heap state. To approach this limitation, Calif used the tried and tested method of Spray and Pray.

Object sizes and padding fix offsets inside region-sized spans. We spray several live writer and reader pairs to give the decoder output several possible neighbors. When its region lands immediately below a writer region, the widened reader produces a larger application response. We discard misses and retry until successful.

One early challenge during exploit development was that the AI would get stuck during heap grooming primitive development. In order to save tokens and time, Calif wrote a small heap tracer and visualizer to give the agent information about the heap state, combined with our expertise on specific cases.

Remote Heap Feng Shui

In order to perform remote grooming, each request data would be sent up until the critical heap allocation would occur. By ensuring the data that was previously sent had a TCP acknowledgement, the exploit would have control over the allocations up to date. Once we were confident our data so far is buffered on the target, we then sent the rest, completing the http request. This allows both heap allocation control and more precise schedule as delivering small network packets is more stable than big ones.



The exact allocation and free order schedule belongs to the targeted application profile, while we are able to control the adding, writer, reader, and decoder pieces.

The current attack is sensitive to network conditions, like RTT and jitter. Low, stable round-trip times are preferred for a more reliable attack. In our controlled environment, we performed this attack over the internet through public IP addresses instead of just assuming perfect laboratory conditions. For the current attack path to achieve optimal stability and reliability in exploitation, Calif turned to physics and utilized money to solve the problem, rather than experimenting with TCP/IP stack configuration. In other words, Calif rented another VM in the same region as our target.

Warm-up, rehearsal, and retries

To summarize the chain so far, we will warm up the Base64 decoder to C2 tier. We can rehearse the exploit run to calibrate timing and responses against the target before causing corruption to occur. Once the required information has been obtained, we can trigger the vulnerability. Incorrect placement could cause the JVM to crash, but in many instances, we can simply try again.

WARM
Run valid decodes until the JIT selects the affected fast path. No memory is corrupted.

REHEARSE
Optional safe round for calibration.

TRIGGER
Send the malformed tail. If the objects line up, the fixed spill enlarges the writer.

DISCLOSE
Continue the writer so chosen bytes enlarge the reader, then collect the response-derived heap bytes.

REPAIR
Use the disclosed originals to restore temporary headers and untouched neighbors when the chain has a repair path.

FINISH
Make the application-specific change and verify it through a live request.

Limitations

Request streaming. Some current grooming profiles need partial request bodies to reach the application. A proxy that waits for the complete body removes their release timing. nginx buffers request bodies by default.

Request buffering is not a general mitigation. Elasticsearch runs from one complete request, so this setting does not stop that chain. After a configured WebSocket upgrade, later frames pass through a tunnel and request-body buffering no longer applies to them.

Application profiles. There is no universal payload. Each chain needs a decoder route, a live writer, a returned reader or another feedback path, a heap layout, and an application action. Every target needs a bespoke profile. The exploit is not a universal point-and-shoot.

PoC

Here is a self-contained proof of concept. It warms the Base64 decoder until the vulnerable intrinsic is used, then decodes the malformed `64 × "A" + "="` input into a 4096-byte array. On an affected JDK it prints `REPRODUCED`: the decoder rejects the input, but 62 bytes (offsets 50 to 111) past the decoded output have already been overwritten. Every write lands inside a real array, so the trigger is safe to run.

```
COPY
import java.nio.charset.StandardCharsets;
import java.util.Arrays;
import java.util.Base64;

/** Bounded standalone trigger: every write stays inside a real 4096-byte array.
*/
public final class Base64Trigger {
    private static volatile int sink;

    public static void main(String[] args) {
        Base64.Decoder decoder = Base64.getDecoder();
        byte[] valid = "A".repeat(64).getBytes(StandardCharsets.US_ASCII);
        byte[] malformed = ("A".repeat(64) +
"=").getBytes(StandardCharsets.US_ASCII);
        byte[] target = new byte[4096];

        for (int i = 0; i < 200_000; i++) {
            sink += decoder.decode(valid, target);
        }

        Arrays.fill(target, (byte) 0x5a);
        boolean rejected = false;
        try {
            decoder.decode(malformed, target);
        } catch (IllegalArgumentException expected) {
            rejected = true;
        }

        int changed = 0;
        int first = -1;
        int last = -1;
        for (int i = 50; i < target.length; i++) {
            if (target[i] != (byte) 0x5a) {
                changed++;
                if (first < 0) first = i;
                last = i;
            }
        }

        System.out.println("runtime=" +
System.getProperty("java.runtime.version"));
        System.out.println("vm=" + System.getProperty("java.vm.name"));
        System.out.printf("rejected=%s changed=%d first=%d last=%d%n",
            rejected, changed, first, last);

        if (!rejected || changed != 62 || first != 50 || last != 111) {
            System.err.println("NOT REPRODUCED");
            System.exit(1);
        }
        System.out.println("REPRODUCED");
    }
}
```

We are holding back the application-specific exploits for now. This write-up covers the vulnerability and the general exploitation technique. The weaponized chains for individual products stay private until everyone has had a chance to upgrade. We will publish them later.

Affected products

Java4Shell affects OpenJDK from version 17.0.6 or newer.

Because they are built from OpenJDK, the following Java distributions are also vulnerable:

- Amazon Corretto
- Eclipse Temurin
- Microsoft Build of OpenJDK
- Azul Zulu
- BellSoft Liberica
- SAP SapMachine
- Red Hat OpenJDK
- Oracle OpenJDK
- Oracle Java SE
- Oracle GraalVM for JDK

According to Oracle, the following Oracle Java SE versions are vulnerable: 8u511b33-perf, 17.0.21.0.3, 21.0.13.0.3, 25.0.5.0.3, 27.0.1.0.3. We thank Oracle for providing this information ahead of publication time.

Because the vulnerable code is hardware-specific, Java4Shell also requires an x86-64 processor with AVX-512 VBMI support. ARM64 CPUs are not affected.

These mainstream CPU families meet the hardware requirements:

- Intel Xeon: Ice Lake, Sapphire Rapids, Emerald Rapids, Granite Rapids
- AMD EPYC and Ryzen: Zen 4 or newer

On Linux, you can check whether your CPU is vulnerable with this command:

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

The output must include all four flags.

Recommendations

OpenJDK released a security advisory and Oracle released a patch at 1:00pm PT on Nov 17, 2026.

All Java users should update immediately.

You can also mitigate Java4Shell by starting OpenJDK with either:

- `-XX:-UseBASE64Intrinsics` to disable hardware-accelerated Base64 encoding and decoding, or
- `-XX:UseAVX=2` to disable AVX-512. Apply this flag only on x86-64 hosts. OpenJDK on ARM doesn't recognize it and will fail to start.

Credits

Research, write-up and exploits by Quang Luong.

Exploit improvement by Jenish Rakholiya.

Artwork by Michal Zalewski.