Exit 137

Why Your Java Service Dies Without an OutOfMemoryError

Chandrakanth Anne | September 21, 2026 | annechandra.com

Your pod restarts. Exit code 137. You open the logs expecting an OutOfMemoryError and find nothing - no stack trace, no heap dump, no warning. You pull up Grafana and heap usage is flat, sitting comfortably around 60% of -Xmx.

Someone suggests raising -Xmx. You do. It dies faster.

I have watched this loop consume a week of a good team's time more than once. It is persistent because it contradicts the mental model most of us carry: that Java memory means the heap.

It does not. The heap is usually the largest part of a JVM's footprint. It is almost never all of it.

What the kernel is actually measuring

When a container is OOM-killed, the kernel is not reading your heap. It is comparing the cgroup's memory usage against its limit - and that number includes every byte the JVM process has resident, not just the Java heap.

Exit 137 is 128 + 9: killed by SIGKILL. The JVM gets no opportunity to write a heap dump, log a message, or run a shutdown hook. That silence is the diagnostic signal. An OutOfMemoryError in the logs and an exit 137 are different failures, and treating them as the same thing is what starts the doubling-Xmx death spiral.

The JVM's memory, in full

Region	Controlled by	Notes
Heap	-Xmx / -Xms	Young and old generation. The only thing a heap dump shows.
Metaspace	-XX:MaxMetaspaceSize	Class metadata. Unbounded by default. Native memory, not heap.
Compressed class space	-XX:CompressedClassSpaceSize	1 GB reserved by default.
Code cache	-XX:ReservedCodeCacheSize	JIT-compiled code; 240 MB default under tiered compilation.
Thread stacks	-Xss	Around 1 MB each on 64-bit Linux. 500 threads is roughly 500 MB of reservations.
Direct byte buffers	-XX:MaxDirectMemorySize	Defaults to roughly -Xmx. Netty, gRPC, Kafka clients and NIO live here.
GC structures	Collector choice	Remembered sets, card tables. Often 5-10% of heap size.
Native libraries / JNI	-	Compression, crypto, native JDBC drivers.
malloc arenas	MALLOC_ARENA_MAX	glibc per-thread arenas hold freed memory the JVM never sees again.

RSS is approximately: committed heap + metaspace + code cache + thread stacks + direct buffers + GC overhead + native libraries + allocator fragmentation

One nuance that matters and is often glossed over: several of these are reserved address space rather than resident memory. CompressedClassSpaceSize reserves 1 GB but commits far less. Thread stacks commit lazily as frames grow. Reserved memory does not count against your cgroup - committed memory does. If you skip this distinction you will panic at pmap output that is entirely normal.

Why -Xmx equal to the container limit is a bug

This is the single most common misconfiguration I encounter.

A 2 GB container with -Xmx2g leaves zero budget for metaspace, thread stacks, code cache, direct buffers, GC structures, or the allocator. The JVM will happily grow the heap toward 2 GB - behaving exactly as instructed - and the kernel will kill it well before it gets there. The heap graph looks healthy right up to the moment the pod vanishes, because from the heap's perspective nothing was ever wrong.

Modern JVMs read cgroup limits by default (UseContainerSupport, on since 8u191 and 10). Use the percentage flags rather than absolutes:

```
-XX:MaxRAMPercentage=70

-XX:InitialRAMPercentage=70
```

Seventy percent is a reasonable starting point for containers of 1 GB or more. Go lower for smaller ones - the non-heap overhead is largely fixed, so it consumes a much larger fraction of a 512 MB container.

Two related surprises worth knowing.

- Legacy Java 8 in containers. Before 8u191, the JVM read host memory and sized the heap to a quarter of it. A container with a 1 GB limit on a 256 GB node would set -Xmx to 64 GB. If you still run older Java 8 anywhere, this is a live incident waiting to happen.

- CPU limits change your garbage collector. With fewer than two available processors, or under roughly 1792 MB of memory, the JVM selects Serial GC. Teams tune CPU limits for cost, unknowingly switch collectors, and then spend days investigating a latency regression that has nothing to do with their code.

Eight OutOfMemoryErrors, and only two are the heap

When you do get an OutOfMemoryError, read the message. It tells you which subsystem failed, and the remediation differs completely.

Message	What it means
Java heap space	Genuine heap exhaustion. Leak, undersizing, or a transient spike.
GC overhead limit exceeded	Over 98% of time in GC recovering under 2% of heap. Heap again - usually the precursor to the above.
Metaspace	Class metadata. Almost always a classloader leak: redeploys, dynamic proxies, bytecode generation.
Compressed class space	Class pointer space exhausted; very high class counts.
Direct buffer memory	Off-heap NIO buffers. Netty or a client library not releasing.
unable to create native thread	OS thread limit or native memory exhaustion. Raising -Xmx makes this worse.
Requested array size exceeds VM limit	A single array beyond the JVM's maximum.
Out of swap space?	A native allocation failed outright.

That sixth one deserves emphasis. 'unable to create native thread' means the process could not get memory for a new thread stack. A larger heap leaves LESS room for thread stacks inside the same container. The instinctive fix is precisely backwards - and I have seen it applied confidently, twice, on the same incident.

A diagnosis order that works

1. Establish which failure you have. OutOfMemoryError in the logs, or exit 137 with silence? Heap problem versus total-footprint problem. Everything downstream depends on this.

2. Measure the gap. Compare container working set against committed heap. In Kubernetes that is container_memory_working_set_bytes against your JVM heap metric. A large and growing delta means your problem is native, and no amount of heap analysis will find it.

3. Turn on Native Memory Tracking. Underused and frequently decisive. Take a baseline, come back an hour later, diff it. The category that grew is your answer. Expect roughly 5-10% overhead, which is usually an acceptable trade during an investigation.

```
-XX:NativeMemoryTracking=summary

jcmd <pid> VM.native_memory summary.diff
```

4. Capture heap dumps that survive. -XX:+HeapDumpOnOutOfMemoryError is necessary but not sufficient in Kubernetes - writing a multi-gigabyte dump takes time and disk, and a pod being killed provides neither. Mount a volume sized for it, and make sure your termination grace period allows the write to finish.

5. For heap leaks, go to the dominator tree. Eclipse MAT's dominator tree finds retention faster than browsing the histogram. Java Flight Recorder is the lower-overhead option for allocation profiling on a live production service.

6. Check MALLOC_ARENA_MAX. glibc creates up to eight arenas per core, each retaining freed memory that never returns to the OS. On a high-core node with a thread-heavy JVM, the fragmentation is substantial. MALLOC_ARENA_MAX=2 is a one-line change that has resolved more container OOMs than it has any right to.

The architectural point

The debugging is worth knowing. The lesson underneath it matters more.

Containers made JVM memory an architecture problem rather than a tuning problem. On a dedicated host with 64 GB of RAM, non-heap overhead was rounding error - nobody had to reason about it. In a 2 GB container it can be a third of your budget, and the parts of it that are not the heap are precisely the parts your dashboards do not show you.

Which leads to the broader habit: know what your observability does not cover. A heap graph is not a memory graph. It was never meant to be. The failure mode here is not that the JVM is mysterious - it is that we monitored the part we understood and inferred the rest, and the gap between those two is exactly where the incident lives.

That pattern is not unique to Java. It is most of what makes production debugging hard.

What is the most misleading dashboard you have debugged against?

Originally published on LinkedIn. Read more at annechandra.com.