
ThreadGlass

# License

*The full repository license is included here so packaged copies of the guide carry the exact license text in a fixed location.*

# Table of Contents

# About This Guide

This guide is written from the current public download page, the public walkthrough page, and the repository implementation. It is meant to help a new user install Thread Glass, run it confidently, and understand the report it produces.

Thread Glass is a command-line diagnostic tool for Java processes. In the current codebase, it attaches to a local JVM by PID, collects a normalized snapshot of the process, analyzes that snapshot, and renders either a concise text report or a structured JSON report. The public walkthrough page reflects that core behavior, and the repository confirms the concrete command surface that exists today.

One important distinction matters for this guide: the repository specs describe some broader product goals such as remote JMX and offline bundle analysis, but the code that currently ships only implements local inspection. Everything below is therefore grounded in the behavior that is actually present now rather than the behavior that may exist later.

# Installation

## Choosing The Right Package

The public download page currently publishes six native packages for Thread Glass version 1.0.0. macOS users can choose either a disk image or an installer package, and each of those is published separately for Intel and Apple Silicon systems. Linux users are given x64 packages in both Debian-style and RPM-style formats. The page does not currently list a Windows package.

Choose the package that matches both your operating system family and your CPU architecture. On macOS, that means deciding whether your machine is Intel or Apple Silicon before downloading. On Linux, that means deciding whether your distribution expects `.deb` packages or `.rpm` packages. If you install the wrong architecture build, the application may fail to start or may never appear as a runnable command. The right time to make that choice is before the download, not after an installer has already failed.

## Installing On macOS

On macOS, begin by deciding both architecture and installer style. Apple Silicon Macs need the `arm64` package; Intel Macs need the `x64` package. Once that is decided, choose between `.pkg` and `.dmg`. The `.pkg` option is the better fit if you want the operating system to walk you through an installer flow. The `.dmg` option is better if you prefer the usual macOS pattern of opening a disk image and installing from there. In both cases, the real goal is the same: finish with a usable Thread Glass application and a working `thread-glass` command in Terminal.

A careful install flow on macOS is straightforward. Download the package that matches your Mac. Open it from your browser downloads list or Downloads folder. Complete the installer or disk-image flow fully before testing anything. Then open a fresh Terminal window rather than reusing one that was already open during installation. That last step matters because shell path changes or application registration may not be visible in an older terminal session.

Your first validation step should be the version command, because it confirms that the application is installed and that the shell can reach the executable. If the command is not found immediately, do not assume the install failed. Some package layouts place the executable inside an application bundle rather than directly on your shell path. In that case, find the executable first, run it directly once, and only then decide whether a path adjustment or installer issue exists.

```
thread-glass version

find /Applications -path '*thread-glass.app/Contents/MacOS/thread-glass' 2>/dev/null
```

If macOS warns that the application came from the internet, clear the security prompt in System Settings and retry. That is an operating-system trust decision, not a Thread Glass-specific runtime error. Treat it as

an installation gate, not as evidence that the application itself is malfunctioning.

## Installing On Debian Or Ubuntu

On Debian-family systems, download the x64 `.deb` package to a known location such as your Downloads folder, then install it with the native package manager. Installing through the platform package manager is preferable to unpacking or improvising because it gives you the expected Linux ownership, file placement, and registration behavior.

After installation, validate in a new shell session with the version command. If the command is not found, check whether the package installed under a standard location such as `/opt` or `/usr`. That is a path-discovery problem, not necessarily a packaging failure. The right workflow is: install, validate, then investigate visibility if needed.

```
sudo apt install ./thread-glass-1.0.0-linux_amd64.deb

thread-glass version
```

## Installing On RPM-Based Linux

On Fedora, Rocky, AlmaLinux, RHEL, openSUSE, and similar systems, use the x64 `.rpm` package. The shape of the process is the same as on Debian-family systems: download first, install with the native package manager, then validate in a new shell session. The important discipline is to let the package manager perform the installation rather than treating the file like a generic archive.

Validation is again simple and deliberate: run the version command first. If it resolves, installation is effectively complete. If it does not, locate the installed executable before making assumptions about a broken release. This avoids wasting time debugging a shell-path issue as though it were an application defect.

```
sudo dnf install ./thread-glass-1.0.0-linux_amd64.rpm

thread-glass version
```

*If the command does not resolve, search common installation roots with `find /opt /usr -name thread-glass -type f 2>/dev/null`.*

# Running Thread Glass

## What The Current CLI Supports

The implemented command surface is intentionally small. The application supports `help`, `version`, and `inspect`. The `inspect` command requires a local PID and optionally accepts `--format json` if you want machine-readable output. If you do not specify a format, the tool renders a text report.

```
thread-glass help

thread-glass inspect --pid 12345

thread-glass inspect --pid 12345 --format json > report.json
```

## Finding The Right PID

Before you run Thread Glass, you need the PID of the Java process you actually want to inspect. The cleanest way to get that is usually `jcmd -l`, because it lists Java process IDs together with the main class or launcher details. If that is not available, `jps -l` is a reasonable fallback. Generic process listings such as `ps` also work, but they are easier to misread when several Java processes are running on the same machine.

```
jcmd -l
```

When you have multiple candidate JVMs, double-check the PID before running an inspection. The report will only be useful if it matches the process you intended to diagnose.

## What Happens During Inspection

When you run `inspect --pid`, Thread Glass attaches to the target JVM, starts a local management agent if needed, reads a set of management beans, maps the results into one immutable snapshot model, runs a simple analyzer against that snapshot, and prints the result. In the current implementation, the collector gathers runtime metadata, operating-system details, memory usage, memory pool data, garbage collection data, thread data, class loading data, and compilation data.

That means the tool is best used as a point-in-time diagnostic snapshot. It is not trying to be a long-running monitor, and it is not trying to replace a full profiler. Its strength is that it can give you a compact JVM health picture quickly, with JSON available when you need to inspect the raw structure more closely.

# How To Read One Detail In Both Formats

The most useful way to learn Thread Glass is to connect one exact text detail to the exact JSON fields behind it. The sections below do that directly. Each one shows the same signal first as terminal output, then as JSON, then explains how to read it and why it matters.

## 1. Process Identity And Capture Time

**Text Output**

```
Target: PID 12345 (ExampleService)
JVM: OpenJDK 64-Bit Server VM 21.0.2
Captured: 2026-04-10T16:00:00Z
```

**JSON Output**

```
{
  "target": {
    "pid": 12345,
    "displayName": "ExampleService",
    "jvmName": "OpenJDK 64-Bit Server VM",
    "jvmVersion": "21.0.2"
  },
  "capturedAt": "2026-04-10T16:00:00Z"
}
```

These two snippets describe the same capture identity. The text version is compressed for fast human scanning. The JSON version breaks the same information into fields you can sort, archive, or compare mechanically. Users should always start here. If the PID, display name, or capture timestamp is wrong, the rest of the report may be accurate for the wrong process, which makes the whole exercise misleading.

The `Target` line in text mode is derived from `snapshot.target`. `Captured` comes from `snapshot.capturedAt`. That means if a teammate forwards only a screenshot of the text output, the matching JSON fields to verify identity are immediately obvious. This is also why `capturedAt` matters operationally: if you collect two reports during an incident, the timestamps tell you whether you are looking at the same failure stage or at a later recovery or degradation phase.

## 2. Overall Summary And Key Points

**Text Output**

```
Summary
- JVM is under elevated pressure
- 182 live threads
- 3 blocked threads
- Heap used 1610612736 / 2147483648 bytes
```

**JSON Output**

```
{
  "analysis": {
    "summary": {
      "headline": "JVM is under elevated pressure",
      "keyPoints": [
        "182 live threads",
        "3 blocked threads",
        "Heap used 1610612736 / 2147483648 bytes"
      ]
    }
  }
}
```

This pair shows the same summary in the two formats. In text mode, the section is already shaped for reading order: headline first, key points underneath. In JSON, the same content lives inside `analysis.summary`. Users should interpret this section as triage guidance, not final proof. It tells you which parts of the report deserve attention next.

Each key point is a compressed sentence that points back to a larger part of the snapshot. A thread-count line points back to `snapshot.threads`. A heap-usage line points back to `snapshot.memory.heap`. This is why summary lines are valuable even though they are short: they reduce the full report into a handful of doors, and then the JSON tells you exactly which room each door opens into.

## 3. Heap Pressure As A Concrete Example

**Text Output**

```
- Heap used 1610612736 / 2147483648 bytes

[MEDIUM][HIGH] Heap pressure is elevated
  Heap usage is above the configured warning threshold.
```

**JSON Output**

```
{
  "memory": {
    "heap": {
      "usedBytes": 1610612736,
      "committedBytes": 1879048192,
      "maxBytes": 2147483648
    }
  },
  "analysis": {
    "findings": [
      {
        "id": "heap-pressure",
        "severity": "medium",
        "confidence": "high",
        "title": "Heap pressure is elevated"
      }
    ]
  }
}
```

This is one of the most important report-reading patterns: a text key point and a text finding both map back to raw memory values in JSON. The text report tells the operator that heap pressure is worth attention. The JSON shows why. `usedBytes` and `maxBytes` are the core numbers. `committedBytes` adds capacity context, showing how much memory the JVM has already reserved from the operating system.

The right way to read this is not simply, 'the tool said heap pressure exists.' The right way is, 'the tool said heap pressure exists, and the raw heap numbers are the evidence behind that statement.' That distinction matters. It teaches users to audit conclusions. When a report says memory is the problem, this is the exact JSON segment that should confirm or weaken that claim.

## 4. Blocked Threads And Contention

**Text Output**

```
- 3 blocked threads

[MEDIUM][MEDIUM] Lock contention is elevated
  Multiple threads are blocked waiting for monitors.
```

**JSON Output**

```
{
  "threads": {
    "threadCount": 182,
    "deadlockedThreadIds": [],
    "threads": [
      {
        "name": "ForkJoinPool.commonPool-worker-3",
        "state": "BLOCKED",
        "blockedTimeMs": 4412,
        "lockName": "java.lang.Object@2f4a1c8b",
        "lockOwnerName": "main"
      }
    ]
  }
}
```

This pair shows how a short text warning about blocked threads expands into actionable thread data in JSON. The text version tells the user the problem category: blocked or contended threads. The JSON version tells the user which thread is blocked, how long it has been blocked, and which lock owner appears to be involved.

When users see blocked-thread or contention language in the text report, the fastest JSON fields to examine are `state`, `blockedTimeMs`, `lockName`, and `lockOwnerName`. Those four fields usually answer the first operational question: is this a brief stall, or is there a real lock bottleneck building? `deadlockedThreadIds` sits above that as a higher-severity shortcut. If that array is non-empty, the problem has moved from contention to a much harder concurrency failure.

## 5. One Finding, One Recommendation, One Evidence Trail

**Text Output**

```
[MEDIUM][HIGH] Heap pressure is elevated
  Heap usage is above the configured warning threshold.
  Recommendation: Inspect allocation rate and retention.
```

**JSON Output**

```
{
  "findings": [
    {
      "id": "heap-pressure",
      "severity": "medium",
      "confidence": "high",
      "title": "Heap pressure is elevated",
      "description": "Heap usage is above the configured warning threshold.",
      "recommendation": "Inspect allocation rate and retention.",
      "evidenceRefs": ["ev-heap-1"]
    }
  ],
  "evidence": [
    {
      "id": "ev-heap-1",
      "kind": "memoryPoolThreshold",
      "details": "Heap usage is 75.0% of max.",
      "data": {
        "usedPercent": 75
      }
    }
  ]
}
```

This is the complete lifecycle of a finding. In text mode, users see a concise human-readable statement with a recommendation. In JSON, that same finding becomes a stable object with an ID, severity, confidence, description, recommendation, and references to evidence. The evidence object then supplies the structured fact payload that the finding was based on.

This relationship is essential if the report is going to be used seriously. A finding is not just an opinion emitted by the tool. It is a claim that can be traced to evidence. `evidenceRefs` is the bridge. Users who want to understand what the tool means by 'heap pressure' should not stop at the finding itself. They should follow the evidence reference and inspect the numeric basis for the claim.

## 6. Scores And Their Matching JSON Fields

**Text Output**

```
Scores
- overall: 19
- memory: 20
- gc: 13
- contention: 25
- threadRisk: 25
- cpu: 10
```

**JSON Output**

```
{
  "scores": {
    "cpuPressureScore": 10,
    "memoryPressureScore": 20,
    "gcPressureScore": 13,
    "contentionScore": 25,
    "threadRiskScore": 25,
    "overallScore": 19
  }
}
```

These are the same numbers in the two output formats. The text report labels them for quick reading. The JSON report preserves the more explicit field names. The operational habit users should develop is to look sideways before looking up. In other words, inspect which component scores are elevated before caring about `overallScore`.

This is what makes the score section useful instead of decorative. A high `overallScore` only says the process looks unhealthy. The component scores say what kind of unhealthy it appears to be. That is why the text report's short labels map cleanly to JSON names such as `memoryPressureScore` and `contentionScore`: the JSON form is what scripts and precise human follow-up should use.

## 7. Supporting Context And Partial Visibility

**Text Output**

```
JVM: OpenJDK 64-Bit Server VM 21.0.2
```

**JSON Output**

```
{
  "runtime": {
    "vmName": "OpenJDK 64-Bit Server VM",
    "vmVendor": "Eclipse Adoptium",
    "vmVersion": "21.0.2"
  },
  "warnings": [
    {
      "code": "partial-collection",
      "message": "Allocated bytes were not available on this target."
    }
  ],
  "jfr": null
}
```

Not every important part of a report is a finding. Some of it is context about what environment was inspected and how complete the collection was. The text report shows the JVM identity directly. The JSON adds the runtime structure, any collection warnings, and optional sections such as `jfr` that may be absent in the current implementation.

Users should read warnings as qualifiers on trust. A report with warnings is not useless, but it is explicitly partial. That means a quiet section elsewhere in the report may reflect missing visibility rather than genuine health. This is one of the most important habits the guide can teach: always check whether the tool had complete enough access to support the conclusion you are about to draw.

## Using The Report Well

If you are diagnosing an incident, start with the JSON output and preserve it. The text output is good for a quick read, but the JSON is the durable artifact. Look for deadlocks first, then blocked-thread patterns, then heap pressure and GC time. If the analyzer says the JVM is degraded but the conclusion feels thin, inspect the raw snapshot and verify the case yourself. The current analyzer is intentionally simple and strongest around deadlocks, blocked threads, heap thresholds, and basic score generation.

If you are comparing two captures over time, focus on changes in thread counts, blocked-thread counts, heap occupancy, GC totals, and the finding list. Those differences usually tell you whether the process is stabilizing, degrading, or stuck in the same failure mode.

# Troubleshooting

If `thread-glass` is not found after installation, open a new terminal and verify where the package placed the executable. If inspection fails, first confirm that the PID is correct and that the target process is actually a live JVM. If the output seems incomplete, inspect `snapshot.capabilities`, `warnings`, and `errors` before deciding that nothing interesting is happening. Incomplete visibility is a normal condition in management tooling, and the report model is designed to preserve that fact explicitly.

For day-to-day use, the most reliable routine is simple: verify the command, identify the PID, capture JSON, and only then read the text summary as a convenience view. That workflow gives you both a quick answer and a durable diagnostic artifact.

## Quick Commands

- `thread-glass version` confirms the installation.
- `thread-glass help` shows the available commands.
- `jcmd -l` helps identify the target JVM PID.
- `thread-glass inspect --pid 12345` renders the text report.
- `thread-glass inspect --pid 12345 --format json > report.json` saves the structured report.