Static Analysis of Redis using Cppcheck

Radu-Constantin Onuțu
Faculty of Mathematics and Computer Science, University of Bucharest

1 Introduction

This report presents the process of finding bugs in Redis with Cppcheck. We begin with an overview of Redis, then introduce Cppcheck and showcase how to use it. Next, we analyse the false positives it generates and highlight genuine issues we identified along with suggestions on how to fix them. In the end we will summarise our results and evaluate the overall effectiveness of Cppcheck on Redis.

2 Repository Overview: Redis

Redis (short for **RE**mote **DI**ctionary **S**erver) is an open-source NoSQL database designed for extreme speed and low-latency workloads. It keeps the entire dataset in RAM (rather than on a disk or SSD) so reads and writes are typically measured in sub-millisecond times. Although its data model is key-value, Redis supports a rich set of value types, including strings, lists, sets, sorted sets, hashes, streams, bitmaps, and HyperLogLogs [1]. This makes it equally adept at acting as a fast cache, real-time analytics engine, or primary data store where rapid access is needed (*Figure 1*).

It was first released in 2009 by Salvatore Sanfilippo. The project has since grown into the world's most widely used NoSQL database [2] (and one of the most popular databases overall [3]).

In practice, Redis is a great option for raw speed requirements (such as for real-time counters or high-throughput caching) while relational databases perform for workload that demands complex queries and robust transactional guarantees.

3 Static Analysis Tool: Cppcheck

Cppcheck is a cross-platform static analysis tool for both C and C++ code, created and led by Daniel Marjamäki [4, 5]. Its core aim is to uncover genuine defects (such as undefined behaviour and other risky constructs) while keeping false positives to an absolute minimum. Unlike many analyzers, Cppcheck handles non-standard syntax common in embedded projects, including compiler extensions and inline assembly, so it can scrutinize code that other tools skip.

The program runs on Windows, POSIX, and other environments. Although its checks are pow-

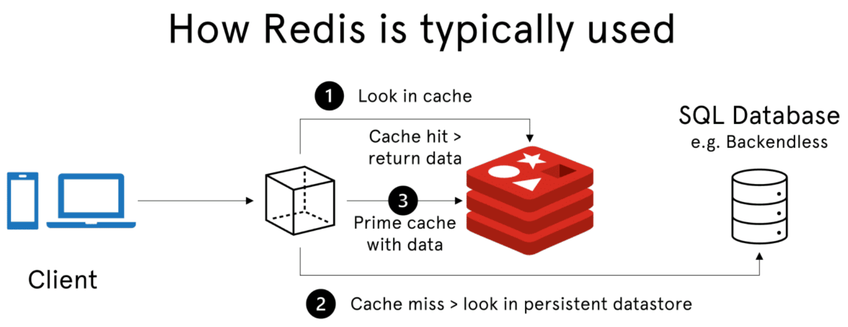


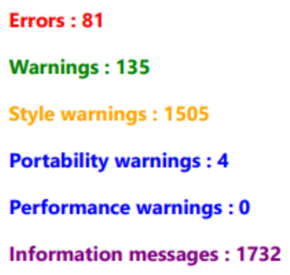
Figure 1: Redis simple use case

erful, they are not exhaustive; some bugs may slip through. Originally called “C++check,” the name was shortened to “Cppcheck,” reflecting its dual focus on C and C++ while keeping the emphasis on practical, low-noise bug detection.

Cppcheck offers a GUI interface that can be especially helpful to less-experienced users. It classifies findings into six categories: errors, warnings, style, performance, portability, and information as shown *Figure 2*. In this report we concentrate chiefly on the errors it detects.

Figure 2: Severities

A full run over the Redis code-base (441 source files) completed in about nine minutes. Cppcheck can also generate a concise statistics summary that lists the number of findings in each severity class, as shown in *Figure 3*.


Errors : 81

Warnings : 135

Style warnings : 1505

Portability warnings : 4

Performance warnings : 0

Information messages : 1732

Figure 3: Statistics on severities

4 Detected Bugs and Suggested Fixes

Before analysing the errors and how to fix them, it is worth noting that Cppcheck reported a considerable number of false positives related to macros. Even macros that are correctly defined and configured can provoke warnings such as:

```
error: There is an unknown macro here somewhere. Configuration is required. If ... is a macro then please configure it. [unknownMacro]
```

Because Cppcheck lets you suppress errors by ids, we suppressed every message tagged `unknownMacro` so that we could focus on the real errors.

Let’s now dive into the more interesting errors based on their CWE (Common Weakness Enumeration) [6] and let’s see suggestions on how to fix them.

- **Error: CWE 401** ("va_list ’_cpy’ used before va_start() was called") - "va_list _cpy;" becomes valid only after it is initialized with either "va_start" or "va_copy()". In the current code the copy is made unconditionally and an early return path may exit the function before "_cpy" is ever given a matching "va_end()".

  Solution :

  ```
  va_list _cpy;
  va_copy(_cpy, ap);
  ```

- **Error: CWE 457** ("Uninitialized struct member: c.err") - In "test_invalid_timeout_errors()" Cppcheck sees a path where the variable "redisContext *c" is never assigned and execution continues to "text_cond...’Invalid timeout specified’".

  Solution :

  ```
      } else {
          fprintf(stderr, "Unsupported connection type %d\n", config.type);
          abort();
      }
  ```

- **Error: CWE 457** ("Uninitialized struct member: node") - "RTREE_GET_CHILD(0)" expects the local pointer "node" to already hold the root of the radix-tree. In the original code "node" is declared but never set before the macro is expanded, so the first statement inside the macro dereferences an indeterminate struct.

  Solution :

  ```
  rtree_node_elm_t *node = rtree_root_node_read(tsdn, rtree, dependent);
  if (node == NULL) {
      return NULL;
  }
  ctx->child[0] = node;
  RTREE_GET_CHILD(0);
  ```

- **Error: CWE 401** ("Memory leak: p") - The test allocates a buffer with "aligned_alloc()", performs a few alignment/contents checks, then just returns. Because "p| is never passed to "free()", Cppcheck warns the error.

  Solution :

  ```
  free(p);
  ```

- **Error: CWE 701** ("Return value of allocation function 'realloc' is not stored.") - "realloc(p, 0)" does free the buffer, but the call still returns a pointer (usually "NULL").

  Solution :

  ```
  p = realloc(p, 0);
  ```

- **Error: CWE 401** ("Common realloc mistake: 'p' nulled but not freed upon failure") - "realloc(p, large0)" overwrites "p" directly. If the reallocation fails and returns "NULL", the original pointer is lost, so the block is leaked.

  Solution :

  ```
  {
      void *newp = realloc(p, large0)
      ;
      assert_ptr_not_null(newp, "
      realloc() failed in arena_decay
       test");
      p = newp;
  }
  ```

- **Error: CWE 415** ("Memory pointed to by 'ptr' is freed twice.") - The unit-test intentionally calls "free(ptr)" twice to make sure jemalloc's debug build aborts on a double-free, but Cppcheck still treats this as a real defect.

- **Error: CWE 682** ("If memory allocation fail: pointer addition with NULL pointer.") - "elem_iter" comes from a heap allocation a few lines above. If that call were ever to return"NULL", the subsequent arithmetic would attempt pointer math on a null pointer.

  Solution :

  ```
  assert_ptr_not_null(elem_iter, "
      calloc() failed in mpsc_queue
      test");
  ```

- **Error: CWE 628** ("Invalid sqrt() argument nr 1. The value is -1 but the valid values are '0.0:'.") - cJSON.c ships its own fallback definition of "NAN" that expands to "sqrt(-1)". Expanding that macro inside the "return (double) NAN;" statement makes static analysis see a call to "sqrt()" with "–1".

5 Conclusion

In this report we showed our efforts to run a static analysis tool over a very popular NoSQL database. Cppcheck generated numerous false positives and we still uncovered several authentic bugs and illustrated a way of fixing them.

References

[1] Last accessed 4 May 2025. URL: https://github.com/redis/redis.

[2] Last accessed 4 May 2025. URL: https://db-engines.com/en/ranking/key-value+store.

[3] Last accessed 4 May 2025. URL: https://db-engines.com/en/ranking.

[4] Last accessed 4 May 2025. URL: http://cppcheck.net/.

[5] Last accessed 4 May 2025. URL: https://github.com/danmar/cppcheck.

[6] Last accessed 4 May 2025. URL: https://cwe.mitre.org/.