Zellic

Prepared for
James Hunsaker
Category Labs, Inc.

Prepared by
Kuilin Li
Avi Weinstock
Zellic

September 8, 2025

Monad

Program Security Assessment - Database

Contents

About Zellic

Zellic is a vulnerability research firm with deep expertise in blockchain security. We specialize in EVM, Move (Aptos and Sui), and Solana as well as Cairo, NEAR, and Cosmos. We review L1s and L2s, cross-chain protocols, wallets and applied cryptography, zero-knowledge circuits, web applications, and more.

Prior to Zellic, we founded the #1 CTF (competitive hacking) team ↗ worldwide in 2020, 2021, and 2023. Our engineers bring a rich set of skills and backgrounds, including cryptography, web security, mobile security, low-level exploitation, and finance. Our background in traditional information security and competitive hacking has enabled us to consistently discover hidden vulnerabilities and develop novel security research, earning us the reputation as the go-to security firm for teams whose rate of innovation outpaces the existing security landscape.

For more on Zellic's ongoing security research initiatives, check out our website zellic.io ↗ and follow @zellic_io ↗ on Twitter. If you are interested in partnering with Zellic, contact us at hello@zellic.io ↗.

1. Overview

1.1. Executive Summary

Zellic conducted a security assessment for Category Labs, Inc. from July 7th, 2025 to September 5th, 2025. The assessment encompassed multiple components, including the compiler, consensus, execution, RPC, networking, and database layers. The review was conducted in parallel, with dedicated teams focusing on distinct portions of the codebase. During this engagement, Zellic reviewed Monad's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.2. Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- Is the database only written and accessed by components authorized to write and access it?
- Does the database provide integrity for data written to it?
- Does the database correctly persist the data written to it consistently with its in-memory representation?
- Are the asynchronous parts of the database deadlock-free?
- Are there any crashes, memory unsafety, or undefined behavior?

1.3. Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- Front-end components
- Infrastructure relating to the project
- Key custody

Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

1.4. Results

During our assessment on the scoped Monad targets, we discovered three findings, all of which were informational in nature.

Breakdown of Finding Impacts

Impact Level	Count
🟥 Critical	0
🟧 High	0
🟨 Medium	0
🟩 Low	0
⬜ Informational	3

2. Introduction

2.1. About Monad

Category Labs, Inc. contributed the following description of Monad:

> The Monad protocol is an L1 blockchain designed to deliver full EVM compatibility with significant performance improvements. On current (testnet) releases, the client developed by Category Labs has been capable of thousands of tps (transactions per second), 400ms block times and 800ms finality with a globally distributed validator set. Monad's performance derives from optimization in several areas:
>
> - MonadBFT for performant, tail-fork-resistant BFT consensus
> - RaptorCast for efficient block transmission
> - Asynchronous Execution for pipelining consensus and execution to raise the time budget for execution
> - Parallel Execution for efficient transaction execution
> - MonadDb for efficient state access
>
> To develop the Monad client software, the engineering team at Category Labs draws upon deep experience from high frequency trading, networking, databases, web3 and academia. For more on Category's ongoing technical work, check out the category.xyz website and follow @category_xyz on Twitter.

2.2. Methodology

During a security assessment, Zellic works through standard phases of security auditing, including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Basic coding mistakes. Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on as necessary. We also perform a cursory review of the code to familiarize ourselves with the targets.

Architecture risks. This encompasses potential hazards originating from the blueprint of a system, which involves its core validation mechanism and other architecturally significant constituents influencing the system's fundamental security attributes, presumptions, trust mode, and design.

Arithmetic issues. This includes but is not limited to integer overflows and underflows, floating-point associativity issues, loss of precision, and unfavorable integer rounding.

Implementation risks. This encompasses risks linked to translating a system's specification into practical code. Constructing a custom system involves developing intricate on-chain and off-chain elements while accommodating the idiosyncrasies and challenges presented by distinct programming languages, frameworks, and execution environments.

Availability. Denial-of-service attacks are another leading issue in custom systems. Issues including but not limited to unhandled panics, unbounded computations, and incorrect error handling can potentially lead to consensus failures.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact. Instead, we assign it on a case-by-case basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

2.3. Scope

The engagement involved a review of the following targets:

Monad Targets

Type	C++
Platform	Monad
Target	monad
Repository	https://github.com/category-labs/monad ↗
Version	`a8ee06639d60236367a83a83a6a0c667c68f90cd`

Programs	`mpt/*`
	`execution/ethereum/db/*`
	`async/*`
	`core/io/*`

Target	monad-bft
Repository	https://github.com/category-labs/monad-bft ↗
Version	`be342260a8875c6d0ada60857017ec093a04b844`

Programs	`monad-triedb/*`
	`monad-triedb-cache/*`
	`monad-triedb-utils/*`
	`monad-wal/*`

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of 46.6 person-weeks. This portion of the assessment was conducted by two consultants over the course of eight calendar weeks.

Contact Information

The following project managers were associated with the engagement:

Jacob Goreski
Engagement Manager
jacob@zellic.io ↗

Chad McDonald
Engagement Manager
chad@zellic.io ↗

Pedro Moura
Engagement Manager
pedro@zellic.io ↗

The following consultants were engaged to conduct the assessment:

Kuilin Li
Engineer
kuilin@zellic.io ↗

Avi Weinstock
Engineer
avi@zellic.io ↗

2.5. Project Timeline

The key dates of the engagement are detailed below.

July 7, 2025	Kick-off call
July 21, 2025	Start of primary review period
September 3, 2025	Commit updated to 2854c7d5
September 8, 2025	End of primary review period

3. Detailed Findings

3.1. Exceptions thrown across an FFI boundary are undefined behavior

Target	monad-triedb/triedb-driver/src/triedb.cpp, monad-triedb/src/lib.rs		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

Exceptions thrown across an FFI boundary from C++ called by Rust (or vice versa, a panic thrown from Rust called by C++) are undefined behavior. Currently, `triedb_open` catches exceptions that may be thrown when constructing a `triedb`, but the other `triedb_*` functions exposed to Rust do not catch exceptions. Additionally, `read_async_callback` and `traverse_callback` do not currently catch panics.

Impact

If an exception or panic is thrown, one runtime's stack unwinding machinery may parse unrelated stack data as its own unwinding metadata, potentially resulting in arbitrary code execution if the stack contains attacker-controlled data.

Recommendations

The `triedb_*` functions in `triedb-driver` should catch exceptions, and `read_async_callback` `traverse_callback` should catch panics.

Remediation

This issue has been acknowledged by Category Labs, Inc., and a fix was implemented in commit `50a648f8` ↗.

This commit replaces thrown exceptions with macros that expand to calls to `monad_assertion_failed`, which calls `abort`, which does not perform stack unwinding.

3.2. FileDb stores all data in a single directory

Target	category/execution/ethereum/db/file_db.cpp		
Category	Coding Mistakes	Severity	Informational
Likelihood	N/A	Impact	Informational

Description

FileDb stores its key-value pairs as files in a single directory instead of in multiple directories.

```
class FileDb::Impl
{
    // ...
    void upsert(char const *const key, std::string_view const value) const
    {
        auto const path = dir_ / key;
        std::stringstream temp_name;
        temp_name << '_' << key << '.' << std::this_thread::get_id();
        auto const temp_path = dir_ / temp_name.str();
        std::ofstream out{
            temp_path, std::ios::out | std::ios::trunc | std::ios::binary};
        MONAD_ASSERT(out);
        out.write(&value[0], static_cast<std::streamsize>(value.size()));
        out.close();
        std::filesystem::rename(temp_path, path);
    }
    // ...
}
```

Since it's used by BlockDb to store one block per file, this leads to a large number of files stored in the same directory.

```
void BlockDb::upsert(uint64_t const num, Block const &block) const
{
    auto const key = std::to_string(num);
    auto const encoded_block = rlp::encode_block(block);
    size_t brotli_size = BrotliEncoderMaxCompressedSize(encoded_block.size());
    MONAD_ASSERT(brotli_size);
    byte_string brotli_buffer;
    brotli_buffer.resize(brotli_size);
    auto const brotli_result = BrotliEncoderCompress(
        BROTLI_DEFAULT_QUALITY,
        BROTLI_DEFAULT_WINDOW,
        BROTLI_MODE_GENERIC,
        encoded_block.size(),
        encoded_block.data(),
        &brotli_size,
        brotli_buffer.data());
    MONAD_ASSERT(brotli_result == BROTLI_TRUE);
    brotli_buffer.resize(brotli_size);
    std::string_view const value{
        reinterpret_cast<char const *>(brotli_buffer.data()),
        brotli_buffer.size()};
    // db_ is FileDb here
    db_.upsert(key.c_str(), value);
}
```

Impact

This may lead to performance problems (such as slow directory enumeration) if the number of files stored grows without bounds, as this isn't a case that filesystems are optimized for.

`BlockDb::upsert` and `FileDb::upsert` are not currently used outside of tests, and `BlockDb` is currently the only user of `FileDb`, so there is no current impact, however, this may change if additional uses of `BlockDb` or `FileDb` are added.

Recommendations

Split the key into chunks (creating a trie structure where each folder is one node), limiting each directory to a few thousand files.

Remediation

This issue has been acknowledged by Category Labs, Inc., and a fix was implemented in commit `c347424e` ↗.

3.3. The completion count returned by `Db::poll` is incorrect under load

Target	category/async/io.cpp		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The `Db::poll` function is declared in category/mpt/db.hpp as a function that is called in order to pump through any async pending DB operations. Its return value comes from its call to `impl_->poll`, which is defined here:

```
virtual size_t poll(bool const blocking, size_t const count) override
{
    return blocking ? aux_.io->poll_blocking(count)
                    : aux_.io->poll_nonblocking(count);
}
```

So the return value comes from the call to either `poll_blocking` or `poll_nonblocking`. These functions are defined in category/async/io.cpp:

```
// Blocks until at least one completion is processed, returning number
// of completions processed.
size_t poll_blocking(size_t count = 1)
{
    size_t n = 0;
    for (; n < count; n++) {
        if (!poll_uring_(n == 0, 0)) {
            break;
        }
    }
    return n;
}

// Never blocks
size_t poll_nonblocking(size_t count = size_t(-1))
{
    size_t n = 0;
    for (; n < count; n++) {
        if (!poll_uring_(false, 0)) {
            break;
        }
    }
    return n;
}
```

According to the comments, this ultimately returns the number of completions processed by the poll. In order to count this, it counts the number of calls to `poll_uring_` that return true.

However, going one more layer into this call, in the definition of `poll_uring_`, we see the following:

```
bool AsyncIO::poll_uring_(bool blocking, unsigned poll_rings_mask)
{
    std::vector<completion_t> completions;
    completions.reserve(
        2 + io_uring_sq_ready(other_ring) +
        ((wr_ring != nullptr) ? io_uring_sq_ready(wr_ring) : 0));
    for (;;) {
        // [...]
        completions.emplace_back(ring, state, std::move(res));
        blocking = false;
    }
    // [...]
    return !completions.empty();
}
```

So one call to `poll_uring_` can actually process multiple completions, and it often will if the database is under heavy load. It returns true if a nonzero amount of completions were processed, so if the caller expects it returning true to mean it completed exactly one completion, then the number of completions will be incorrect.

Impact

Although the comments indicate that the `Db::poll` function — which is part of the external interface of the MPT DB and can be called by other modules — returns the number of completions that it processes during the poll, it actually undercounts the number of completions.

This does not currently have any impact, because `Db::poll` is currently only used in testing code that does not read the return value. However, if future code relies on this interface, then it may behave unexpectedly under load.

Recommendations

We recommend changing the return type of `poll_uring_` so that it can return `completions.size()` instead of `!completions.empty()`. Then, the `poll_blocking` and `poll_nonblocking` functions can accumulate this quantity to return the correct number.

Alternatively, if the number of completions processed will not be used by any consumers of the MPT DB, then these can all return booleans to signify if any number of completions were processed.

Remediation

This issue has been acknowledged by Category Labs, Inc., and a fix was implemented in commit `862155ce` ↗.

4. System Design

This provides a description of the high-level components of the system and how they interact, including details like a function's externally controllable inputs and how an attacker could leverage each input to cause harm or which invariants or constraints of the system are critical and must always be upheld.

Not all components in the audit scope may have been modeled. The absence of a component in this section does not necessarily suggest that it is safe.

4.1. Database design

Architecture

The main database, TrieDb, stores key-value pairs in a Merkle-Patricia trie, allowing versions of the data to be identified by a root hash. The root hash also ensures that if any of the underlying data is modified, the root hash will change, providing data integrity. Data is persisted in chunks allocated by the storage pool from an underlying storage medium (currently either a file or block device). The underlying I/O is performed with the `io_uring` APIs. The database is threaded, with worker threads that process requests to read and modify the trie, and has a read-only mode whose worker threads do not modify the trie.

Data is also stored separately from the main database by `BlockDb`, which stores blocks in files, and `WALogger`, which stores a stream of serialized events to files.

Attack surface

The execution component writes to the database by providing linked lists of key-value pairs to insert or replace. As the execution component validates data before committing it to the database, this ensures that writes to the database are authorized. The Rust bindings allow other components to read from the database but not modify it. Since the data written to the database is the chain state, which is public, no additional access control is needed for reads.

Risks

Although the majority of the attack surface that the database faces is handled by the correctness of the execution layer's calls into the database, there are other risks to be noted for this component.

Of course, because the database implementation runs on physical machines, any single node can be compromised by an attacker or a bug affecting the integrity of the machine itself. These risks are mitigated by the consensus mechanisms that live in the higher levels of abstraction, making the network of nodes more reliable than any single node, under the assumption that such single-node failures are statistically independent. So, the design of the database assumes that there are no malicious processes, users, hardware components, etc, which is a reasonable assumption based on the typical execution environment.

5. Assessment Results

During our assessment on the scoped Monad targets, we discovered three findings, all of which were informational in nature.

5.1. Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the version reviewed during our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.