Zellic

Aug 7, 2025

Prepared for
James Hunsaker
Category Labs, Inc.

Prepared by
Avi Weinstock
Ayaz Mammadov
Bryce Casaje
Can Boluk
Chongyu Lv
Filippo Cremonase
Jinseo Kim
Kuilin Li
Nan Wang
Ziling Chen
Zellic

Monad

Program Security Assessment

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. 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:

- Does the VM's sandbox effectively prevent escapes to the host system?
- Does the compilation process suffer from infinite loops or unexpected overcomputation?
- Do code-integrity mechanisms prevent unauthorized tampering or replacement of compiled code or VM bytecode?
- Are robust memory safety mechanisms in place to mitigate vulnerabilities like buffer overflows, use-after-free, and other undefined behavior?
- Is thorough input validation implemented to handle untrusted or malicious input, preventing crashes or code injection?

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 one finding, which was informational in nature.

Additionally, Zellic recorded its notes and observations from the assessment for the benefit of Category Labs, Inc. in the Discussion section (4. ↗).

Breakdown of Finding Impacts

Impact Level	Count
🟥 Critical	0
🟧 High	0
🟨 Medium	0
🟩 Low	0
⬜ Informational	1

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.

Finally, Zellic provides a list of miscellaneous observations that do not have security impact or are not directly related to the scoped targets itself. These observations — found in the Discussion (4. ↗) section of the document — may include suggestions for improving the codebase, or general recommendations, but do not necessarily convey that we suggest a code change.

2.3. Scope

The engagement involved a review of the following targets:

Monad Targets

Type	C++
Platform	Monad
Target	monad-compiler
Repository	https://github.com/category-labs/monad-compiler ↗
Version	`f65e528e0ec1442afda50f3c8f7c06a5c7701e25`

Programs	`compiler/src/monad/vm/compiler/ir/basic_blocks.cpp`
	`compiler/src/monad/vm/compiler/ir/basic_blocks.hpp`
	`compiler/src/monad/vm/compiler/ir/instruction.hpp`
	`compiler/src/monad/vm/compiler/ir/local_stacks.cpp`
	`compiler/src/monad/vm/compiler/ir/local_stacks.hpp`
	`compiler/src/monad/vm/compiler/ir/poly_typed.cpp`
	`compiler/src/monad/vm/compiler/ir/poly_typed.hpp`
	`compiler/src/monad/vm/compiler/ir/x86.hpp`
	`compiler/src/monad/vm/compiler/ir/x86.cpp`
	`compiler/src/monad/vm/compiler/types.hpp`
	`compiler/src/monad/vm/compiler/ir/x86/*`
	`core/*`
	`evm/*`
	`fuzzing/*`
	`interpreter/*`
	`runtime/*`
	`utils/*`
	`vm/*`

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of 46.6 person-weeks. The assessment was conducted by ten consultants over the course of 9 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:

Bryce Casaje
Engineer
bryce@zellic.io ↗

Can Boluk
Engineer
can.boluk@zellic.io ↗

Ziling Chen
Engineer
ziling@zellic.io ↗

Filippo Cremonase
Engineer
fcremo@zellic.io ↗

Jinseo Kim
Engineer
jinseo@zellic.io ↗

Kuilin Li
Engineer
kuilin@zellic.io ↗

Chongyu Lv
Engineer
chongyu@zellic.io ↗

Ayaz Mammadov
Engineer
ayaz@zellic.io ↗

Nan Wang
Engineer
nan@zellic.io ↗

Avi Weinstock
Engineer
avi@zellic.io ↗

2.5. Project Timeline

July 7, 2025	Kick-off call
July 7, 2025	Start of primary review period
September 5, 2025	End of primary review period

3. Detailed Findings

3.1. Incorrect usage of `evmc_flags`

Target	Compiler		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The `Environment::evmc_flags` variable of type `std::uint32_t` is a bitfield that is supposed to be populated using values from the evmone `evmc_flags` enum.

However, Monad code assumes that `Environment::evmc_flags` is either zero or `EVMC_STATIC`. This is unsound, as with the Pectra upgrade, the `evmc_flags` enum supports an additional value `EVMC_DELEGATED`:

```
enum evmc_flags
{
    EVMC_STATIC = 1,    /**< Static call mode. */
    EVMC_DELEGATED = 2 /**< Delegated call mode (EIP-7702). Valid since
    Prague. */
};
```

In several code locations, `Environment::evmc_flags` is either compared with or assigned the `EVMC_STATIC` value directly instead of using bitwise operations to test/set only the corresponding bit.

For instance, the `create_impl` function, which handles the `CREATE` opcode, executes this code to revert the transaction if the opcode is executed in a `STATICCALL` context:

```
if (MONAD_VM_UNLIKELY(ctx->env.evmc_flags == EVMC_STATIC)) {
    ctx->exit(StatusCode::Error);
}
```

This check is not correct, as it should be `ctx->env.evmc_flags & EVMC_STATIC`.

Other code locations containing this programming error include `log_impl`, `selfdestruct`, `sstore`, and `tstore`.

Impact

The code revision under review at the time of writing does not yet support the Pectra/Prague EVM revision, specifically EIP-7702. The `EVMC_DELEGATED` flag can only be set after these revisions; therefore, while not future-proof, the code technically behaves correctly.

Recommendations

Use appropriate bitwise operations when testing or setting one of the values of `evmc_flags`.

Remediation

This issue has been acknowledged by Category Labs, Inc., and fixes were implemented in the following commits:

- `aa18ecd2` ↗
- `36d49922` ↗

4. Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment. These discussion notes are not necessarily security related and do not convey that we are suggesting a code change.

4.1. Suboptimal operand selection

Binary operations on stack elements whose values are allocated in general registers require emitting pairwise operations on each of the four legs.

This allows the compiler to implement a peephole optimization for some operations (including `ADD`, `SUB`, `AND`, `OR`, `XOR`), where if one of the two operands is known, codegen for some of the legs could be skipped in case the known operand value turns the suboperation into the identity operation.

Operands for commutative binary operations are ranked and reordered to maximize the likelihood that this optimization can be applied. However, the ranking was found to be suboptimal for specific edge cases. While this is not a security issue, the team improved the routine ranking operands in commit `273c1c1b` ↗.

The improved ranking tends to prefer using literal values as a source operand, which is the operand analyzed to determine whether codegen for a given leg can be skipped.

5. 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.

5.1. Compiler design

This section briefly documents some of the major design choices of the Monad compiler, including the triggering of compilation, the compilation process, optimizations, hardening considerations, and the asynchronous compilation architecture.

Triggering compilation

Contracts deployed on Monad are not immediately compiled. Initially, contracts are executed using a regular interpreter. The amount of gas consumed by a contract is recorded, and once a certain gas threshold (proportional to the bytecode size) is reached, the contract is scheduled for compilation.

Compiled contracts are cached in memory and only recompiled if executed for targeting a different EVM revision due to differences in the EVM behavior across revisions.

Compilation process

Monad implements a compiler for EVM bytecode, which targets x86-64 processors with AVX2 support. Compilation happens in a straightforward two-step pipeline. The compiler is entirely ahead of time, with no speculation. Most EVM instructions are emitted directly as a sequence of x86-64 instructions, with exceptions for complex instructions that call into C++ support functions.

First pass: Basic-block analysis

The first pass performs an analysis on the bytecode to identify valid jump targets (`JUMPDEST` markers) and basic blocks. This pass also computes for every block

- the minimum stack height required,
- the maximum stack height increase, and
- the stack height difference at the end of the block.

Second pass: Basic-block machine code emission

The second pass then iterates over each identified basic block and emits the corresponding code. Each basic block starts with a prologue, which performs stack overflow/underflow checks and charges for the static gas used by all the instructions in the block. Checks ensuring there is remaining gas are deferred when possible.

Each EVM instruction contained in the basic block is directly and individually translated to x86-64 machine code.

Memory and registers layout

EVM stack elements

The compiler maintains bookkeeping of the state of the EVM stack as the instructions of a basic block are executed. The number of elements added and removed from the EVM stack is statically known for each opcode, allowing the compiler to track how many elements are present on the stack (relative to its unknown starting amount) at any given moment.

Stack elements can be physically allocated to various physical locations: general-purpose registers, AVX2 registers, and/or in a memory location on the x86 machine stack. Literal values known at compile time are also treated as a special kind of location.

The compiler aggregates three sets of four general-purpose registers to create three general-purpose–allocated 256-bit registers, which can hold EVM values: `(r12-r15)`, `(r8-r11)`, and `(rcx, rsi, rdx, rdi)`. The `ymm0-ymm15` range of 256-bit AVX2 registers is also used for storing EVM stack elements.

The runtime location of EVM stack-elements values changes as the compiler needs to relocate values to free up registers so they can be used to store operands. Spilling from a register to memory is avoided if possible.

All live stack elements are allocated to at least one location at all times (otherwise, their value would be lost). Stack elements that represent the output of a comparison are a notable exception to this.

In general, at the end of a basic block, the values of the EVM stack elements are copied to a canonical location on the machine stack, which matches their offset in the EVM stack. Therefore, at the beginning of every basic block, the EVM stack elements received as arguments are located at a predictable location (with respect to the machine stack pointer).

An exception is made for blocks that can fall through to a block with a single predecessor (i.e., blocks terminating with a `JUMPI` conditional jump and falling through to a block that does not begin with a `JUMPDEST` marker). In this case, the fall-through block is guaranteed to inherit the abstract EVM stack state of the predecessor, and therefore the stack-state bookkeeping is reused without the need to copy EVM stack-element values to a standardized location.

Physical memory allocation

The physical memory regions used to store the EVM stack and EVM memory are allocated by two instances of a simple ad hoc caching allocator.

The stack is bounded in size and is only allocated once, as a 32kB allocation. EVM memory can grow dynamically and is allocated in 4KB chunks. The allocator recycles previous allocations, putting them in a freelist on release.

Optimizations and other performance considerations

The Monad compiler is required to be low latency and have consistent performance characteristics. For this reason, it does not make use of a complex IR or sophisticated analyses, and it does not employ advanced inter–basic-block or inter-instruction optimizations. While this certainly leads to suboptimal code generation, the simpler design lowers the risk of miscompilations and increases the consistency of the compiler performance, which lowers the risk of denial-of-service (DOS) issues.

Several more narrowly scoped optimizations are employed, including constant folding, careful selection of the physical location of the operands, use of the AVX2 instruction set, optimized handwritten assembly routines for large multiplications, many ad hoc peephole optimizations for arithmetic and logic operations, and the deferred-comparison mechanism.

Deferred comparisons

EVM instructions that have a binary outcome can often be compiled using a sequence of x86-64 instructions that ultimately populate the flag register with the result of the EVM instruction execution.

The value of EVM stack elements representing these binary outcomes is not immediately copied in a general-purpose or AVX register or in a machine-stack memory location. Instead, the compiler pushes a special stack element and marks the stack as containing a deferred comparison. The deferred-comparison element has no regular location, as its value is contained in the processor flag register.

The compiler must not emit any instruction affecting the flag register and cannot use the deferred-comparison stack element without first discharging its value into a general-purpose, AVX, or machine-stack location.

Deferred gas checks

The static gas cost for all the EVM instructions in a block is charged only once at the beginning of the block, by decreasing a global context variable keeping track of the remaining available gas. This optimization allows avoiding updating the gas counter for each instruction, only requiring additional work to account for instructions that incur a dynamic gas cost as well.

A check to determine whether the static gas cost has exceeded the remaining gas budget for the transaction is always inserted at the beginning of a basic block if it starts with a `JUMPDEST` marker (jump target basic block, JTBB). This greatly reduces the likelihood of the most severe DOS scenarios where gas is not accounted for a loop of basic blocks, since control-flow instructions (needed to create a loop) are only allowed to transfer control to a `JUMPDEST` target for their non–fall-through path.

If a basic block does not start with a `JUMPDEST` marker, the compiler still emits instructions to update the global remaining gas context variable, but it defers checks for gas underflows until either a JTBB is encountered or until it has emitted a sequence of basic blocks that consume a set threshold amount of gas (currently 1,000).

Instructions that incur dynamic gas costs independently charge their cost when they are executed.

Hardening considerations

The compiled code is executed in the address space of the validator binary without sandboxing. Compilation is entirely deterministic and does not employ common hardening techniques such as layout randomization, constant blinding, and control-flow integrity.

The lack of hardening techniques increases the likelihood of possible compiler bugs to be practically exploitable, posing a risk for the network availability and integrity.

Asynchronous compilation architecture

This subsection documents the asynchronous compilation architecture, including the producer-consumer model, concurrent infrastructure, controlled race conditions, batch processing and hybrid synchronization, and compilation result caching as well as performance characteristics and trade-offs.

Producer-consumer model

The Monad compiler implements a sophisticated asynchronous compilation system that decouples compilation from execution through a producer-consumer architecture. The main execution threads act as producers, submitting compilation jobs without blocking, while a dedicated background compiler thread serves as the consumer, continuously processing the compilation queue.

This separation ensures that transaction execution is never blocked by compilation overhead, directly supporting Monad's high-throughput objectives. The architecture enables true parallelization where current transactions can be executed while future contracts are being compiled in the background, creating an efficient pipeline processing model.

Concurrent infrastructure

The asynchronous system is built on Intel Threading Building Blocks (TBB), providing lock-free concurrent data structures that eliminate traditional mutex-related performance bottlenecks. The core components include the following:

- **Concurrent hash map** — stores compilation parameters indexed by code hash, enabling efficient job deduplication
- **Concurrent queue** — manages the order of compilation tasks, supporting multiple producers and a single consumer
- **Lock-free operations** — all hot-path operations avoid acquiring locks, ensuring consistent low-latency performance

Controlled race conditions

The system deliberately employs a controlled-race-conditions strategy for task-limit enforcement. Rather than using strict locking mechanisms that would introduce latency in the critical path, the compiler accepts that the number of queued compilation jobs may temporarily exceed the soft limit when multiple threads submit jobs concurrently.

This design trade-off embodies the principle of approximate control over precise blocking in high-performance systems. The temporary overage is bounded by the number of concurrent execution threads and is considered acceptable because the memory overhead of queued compilation jobs is asymptotically equivalent to the memory usage of concurrently executing contracts.

Batch processing and hybrid synchronization

The background compiler thread employs a hybrid synchronization mechanism combining condition variable notifications with one-millisecond timeout polling. This time-out interval is carefully calibrated to roughly match the compilation time of typical smart contracts, achieving optimal balance between responsiveness and CPU efficiency.

When awakened, the thread processes all queued compilation tasks in a single batch, maximizing the efficiency of thread context switches and minimizing system-call overhead.

Compilation result caching

The system implements a sophisticated multi-tier caching strategy centered around VarcodeCache, which employs a weighted least recently used (LRU) eviction policy. Cache weights are dynamically adjusted based on bytecode size, ensuring rational memory-utilization patterns.

Smart pointer-based lifetime management enables safe sharing of compilation results across multiple execution contexts. The code hash-based deduplication mechanism ensures that identical EVM bytecode is never compiled redundantly, even under high-concurrency scenarios with potential race conditions.

Performance characteristics and trade-offs

The asynchronous compilation architecture delivers several key performance advantages:

- **Zero-lock hot path.** The main execution threads submit compilation jobs without acquiring any locks.
- **High concurrent scalability.** The TBB-based lock-free design supports arbitrary numbers of concurrent job submitters.
- **Intelligent resource management.**The multi-tier caching and weighted LRU policies optimize memory efficiency.
- **Compilation result reuse.** The hash-based deduplication eliminates redundant compilation overhead.

However, the design also involves intentional trade-offs:

- **Approximate control precision.** The task count limits may be temporarily exceeded to avoid lock contention.
- **Polling overhead.** One-millisecond periodic polling consumes modest CPU resources while ensuring responsiveness.
- **Architectural complexity.** The asynchronous design increases overall system complexity, potentially complicating debugging and maintenance.

6. Assessment Results

During our assessment on the scoped Monad targets, we discovered one finding, which was informational in nature.

6.1. 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.