Zellic

August 15, 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:

- Can attackers exploit RPC interfaces like `eth_call` to compromise execution integrity?
- Are there vulnerabilities in staking-precompile contract implementations?
- Can malformed EVM instructions (`CALL`, `DELEGATECALL`, `CREATE`, `CREATE2`) or improper revert handling lead to execution failures?
- Are there memory safety issues or encoding/decoding asymmetries in RLP processing?
- Can malformed transactions or blocks bypass input validation to cause consensus splits?

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 12 findings. No critical issues were found. One finding was of high impact, five were of medium impact, one was of low impact, and the remaining findings were informational in nature.

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	1
Medium	5
Low	1
Informational	5

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	Rust, C++
Platform	Monad

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

Target	monad
Repository	https://github.com/category-labs/monad ↗
Version	`1a7f9476081abc734fc6fa359698c3b8f9806576`
Programs	`{*.cpp,*.c}` `{*.hpp,*.h}`

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 eleven consultants over the course of nine 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:

Can Boluk
Engineer
can.boluk@zellic.io

Bryce Casaje
Engineer
bryce@zellic.io

Ziling Chen
Engineer
ziling@zellic.io

Filippo Cremonese
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

The key dates of the engagement are detailed below.

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. Out-of-bounds write in event-recorder truncated-payload-size calculation

Target	category/execution/ethereum/event/exec_event_recorder.cpp		
Category	Coding Mistakes	**Severity**	High
Likelihood	Medium	**Impact**	High

Description

In the `ExecutionEventRecorder::setup_record_error_event` function, when handling `MONAD_EVENT_RECORD_ERROR_OVERFLOW_4GB` and `MONAD_EVENT_RECORD_ERROR_OVERFLOW_EXPIRE` error types, there is an out-of-bounds (OOB) write vulnerability. The issue occurs due to logic errors in calculating truncated payload size and available buffer space.

```
case MONAD_EVENT_RECORD_ERROR_OVERFLOW_EXPIRE:
    error_payload->truncated_payload_size = RECORD_ERROR_TRUNCATED_SIZE;
    {
        size_t residual_size = error_payload->truncated_payload_size; //
    Error: uses 8192
        void *p = payload_buf + sizeof *error_payload + header_payload_size;
        for (std::span<std::byte const> buf : trailing_payload_bufs) {
            size_t const copy_len = std::min(residual_size, size(buf));
            p = mempcpy(p, data(buf), copy_len);
            residual_size -= copy_len;
            if (residual_size == 0) {
                break;
            }
        }
    }
    break;
```

The total size of `payload_buf` is `RECORD_ERROR_TRUNCATED_SIZE` (8,192 bytes), but the actual payload layout is as follows:

```
.---------------.----------------------.--------------.
| *error_payload | event header (type T) | truncated VLT |
'---------------'----------------------'--------------'
```

In the code, `error_payload->truncated_payload_size` is set to the full `RECORD_ERROR_TRUNCATED_SIZE` (8,192), but this value should represent the actual size of the truncated payload, excluding the `error_payload` structure itself.

The copy starting position `p` is already offset by `sizeof *error_payload + header_payload_size`, but `residual_size` still uses `8192`. The actual available space should be `8192 - sizeof(*error_payload) - header_payload_size`, which causes `mempcpy` to potentially write beyond the buffer boundary.

Impact

This OOB write vulnerability could be maliciously exploited to compromise memory integrity, leading to execution-layer state anomalies.

Recommendations

Fix the calculation logic for truncated payload size to ensure `residual_size` correctly reflects the actual available buffer space and prevents OOB writes.

Remediation

This issue has been acknowledged by Category Labs, Inc., and a fix was implemented in commit `e43cec2e` ↗.

3.2. Conflict between special auth_address and the sentinel/empty of the linked list

Target	category/execution/monad/staking/staking_contract.cpp		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

In the `precompile_add_validator` function, any `auth_address` from the input is accepted, with no validation performed for reserved values.

```
auto const auth_address =
        unaligned_load<Address>(consume_bytes(reader,
    sizeof(Address)).data());
```

In the `delegate` function, `auth_address` is inserted into the doubly linked list (where `delegators` and `validators` form an adjacency list for each other) as a key. The linked list uses all `FF`s as the `sentinel` and all `0`s as the `empty`.

```
linked_list_insert(val_id, address); // validator => List[Delegator]
linked_list_insert(address, val_id); // delegator => List[Validator]
......
static constexpr Ptr sentinel()
{
    return Ptr{{0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
                0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
                0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF}};
}

static constexpr Ptr empty()
{
    return Ptr{};
}
```

If the `auth_address` is all `FF`s, it will conflict with the `sentinel`; if it is all `0`s, it will conflict with the `empty`. This will corrupt the linked-list structure, with a typical symptom being the formation of a self-loop in the sentinel node.

Impact

First, it may cause damage to the linked-list structure: abnormalities in iteration, insertion, and deletion, which may trigger self-loops, broken links, and so on. Secondly, it will cause unnecessary memory and computing power overhead; due to the existence of self-loops, the number of unnecessary cycles will increase when traversing the linked list.

Recommendations

Validate the `auth_address` or set the `auth_address` to `msg.sender`.

Remediation

This issue has been acknowledged by Category Labs, Inc., and a fix was implemented in commit `2854c7d5` ↗.

3.3. Contradictory input validation in `precompile_get_withdrawal_request`

Target	category/execution/monad/staking/staking_contract.cpp		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	High	**Impact**	Medium

Description

In the `precompile_get_withdrawal_request` function of category/execution/monad/staking/staking_contract.cpp, there are contradictory input-validation checks that make it impossible for any input to pass validation:

```
Result<byte_string> StakingContract::precompile_get_withdrawal_request(
    byte_string_view const input, evmc_address const &,
    evmc_uint256be const &)
{
    if (MONAD_UNLIKELY(!input.empty())) {
        return StakeError::InvalidInput;
    }
    constexpr size_t MESSAGE_SIZE = sizeof(u64_be) /* validator id */ +
                                    sizeof(Address) /* delegator */ +
                                    sizeof(uint8_t) /* withdrawal id */;
    if (MONAD_UNLIKELY(input.size() != MESSAGE_SIZE)) {
        return StakeError::InvalidInput;
    }
    ......
}
```

The function contains two mutually exclusive validation checks:

1. First check — returns error if input is _not_ empty (`!input.empty()`).
2. Second check — returns error if input size is _not_ equal to `MESSAGE_SIZE`.

These conditions cannot be satisfied simultaneously — if the input is empty, its size is `0` and cannot equal `MESSAGE_SIZE`. If the input size equals `MESSAGE_SIZE`, then the input cannot be empty.

Impact

The function will always return `StakeError::InvalidInput` regardless of input, making this staking functionality completely unavailable.

Recommendations

The first validation check should be corrected to reject empty inputs instead of nonempty inputs by removing the negation operator.

Remediation

This issue has been acknowledged by Category Labs, Inc., and a fix was implemented in commit `341ddc31` ↗.

3.4. Incorrect `memcmp` length causing block-ID matching vulnerability

Target	category/execution/ethereum/event/exec_iter_help_inline.h		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

In the `monad_exec_ring_block_id_matches` function, the `MONAD_EXEC_BLOCK_FINALIZED` branch uses an incorrect comparison length for `memcmp`:

```
inline bool monad_exec_ring_block_id_matches(
    struct monad_event_ring const *event_ring,
    struct monad_event_descriptor const *event,
    monad_c_bytes32 const *block_id)
{
    ......

    switch (event->event_type) {
    case MONAD_EXEC_BLOCK_START:
        tag_matches = memcmp(
                          block_id,
                          ((struct monad_exec_block_start const *)payload)
                              ->block_tag.id.bytes,
                          sizeof *block_id) == 0;
        break;

    case MONAD_EXEC_BLOCK_QC:
        tag_matches = memcmp(
                          block_id,
                          ((struct monad_exec_block_qc const *)payload)
                              ->block_tag.id.bytes,
                          sizeof *block_id) == 0;
        break;

    case MONAD_EXEC_BLOCK_FINALIZED:
        tag_matches =
            memcmp(
                block_id,
                ((struct monad_exec_block_tag const *)payload)->id.bytes,
                sizeof &block_id) == 0; // Error: should use sizeof *block_id
        break;
```

```
    default:
        return false;
    }

    return tag_matches && monad_event_ring_payload_check(event_ring, event);
}
```

The issue is using `sizeof &block_id` instead of `sizeof *block_id`:

- `sizeof *block_id` = `sizeof(bytes32)` = 32 bytes
- `sizeof &block_id` = 8 bytes (on 64-bit machines)

This results in comparing only the first eight bytes, causing different block IDs with the same prefix to be incorrectly identified as identical.

Impact

Incorrect block-ID matching may cause incorrect block event associations, affecting execution layer correctness and consensus safety.

Recommendations

Change `sizeof &block_id` to `sizeof *block_id` to ensure complete 32-byte block-ID comparison.

Remediation

This issue has been acknowledged by Category Labs, Inc., and a fix was implemented in commit d92e78d1 ↗.

3.5. Incorrect decoding type for `base_fee_per_gas` causes decoding failure

Target	category/execution/monad/core/rlp/monad_block_rlp.cpp		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

Category Labs, Inc. was aware of this issue at the time of the audit, and we confirmed it independently. As it was present in the reviewed commit, we've documented it here for completeness.

In the category/execution/monad/core/rlp/monad_block_rlp.cpp file, the `base_fee_per_gas` field in the `BlockHeader` struct is defined as the `uint256_t` type:

```
struct BlockHeader
{
    ...
    std::optional<uint256_t> base_fee_per_gas{std::nullopt};
    std::optional<bytes32_t> withdrawals_root{std::nullopt};
    ...
};
```

During encoding, it correctly uses the `uint256_t` type:

```
byte_string encode_block_header(BlockHeader const &block_header)
{
    ...
    if (block_header.base_fee_per_gas.has_value()) {
        encoded_block_header +=
            encode_unsigned(block_header.base_fee_per_gas.value());
    }
    ...
}
```

However, in the decoding function `decode_execution_inputs`, this field is incorrectly decoded as the `uint64_t` type:

```
Result<BlockHeader> decode_execution_inputs(byte_string_view &enc)
{
    BlockHeader header;
    ...
    BOOST_OUTCOME_TRY(
        header.base_fee_per_gas, decode_unsigned<uint64_t>(payload));
    ...
}
```

When the value of `base_fee_per_gas` exceeds the range of `uint64_t`, the decoding process will fail, causing the inability to properly process blocks containing large base fees. This is inconsistent with Ethereum's Geth implementation, which also uses the `uint256_t` type.

Impact

Block-decoding failure when the base fee exceeds the `uint64_t` range may cause nodes to fail synchronization or processing certain blocks.

Recommendations

Change the decoding type for the `base_fee_per_gas` field from `uint64_t` to `uint256_t` in the decoding function to maintain type consistency.

Remediation

This issue has been acknowledged by Category Labs, Inc., and a fix was implemented in commit `bfc5ea71` ↗.

3.6. RPC denial of service caused by `eth_call` `STORAGE_OVERRIDE` functionality

Target	category/execution/ethereum/state3/state.hpp		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	High	**Impact**	Medium

Description

Category Labs, Inc. was aware of this issue at the time of the audit, and we confirmed it independently. As it was present in the reviewed commit, we've documented it here for completeness.

The `STORAGE_OVERRIDE` parameter is an important feature of the `eth_call` RPC interface that allows callers to temporarily modify blockchain state when executing simulated transactions without affecting the actual on-chain state.

Users can arbitrarily set account balances through the `STORAGE_OVERRIDE` parameter to perform transfer transactions. For example, they can set a target account's balance to the maximum `uint256` value then construct a transfer transaction to that account.

Since the target account's balance has already reached the maximum `uint256` value, any transfer to that account will cause a balance overflow, triggering an assertion failure in the `add_to_balance` function.

```
void add_to_balance(Address const &address, uint256_t const &delta)
{
    auto &account_state = current_account_state(address);
    auto &account = account_state.account_;
    if (MONAD_UNLIKELY(!account.has_value())) {
        account = Account{.incarnation = incarnation_};
    }

    MONAD_ASSERT_THROW( // <-------
        std::numeric_limits<uint256_t>::max() - delta >=
            account.value().balance,
        "balance overflow");

    account.value().balance += delta;
    account_state.touch();
}
```

Impact

This causes RPC denial of service, though it will not crash the entire node.

Recommendations

For exceptions such as balance overflow, return execution failure instead of performing assertions.

Remediation

TBD

3.7. The `eth_call` implementation's high-gas-pool resource exhaustion due to lack of execution time-out

Target	category/rpc/eth_call.cpp		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Medium	**Impact**	Low

Description

Category Labs, Inc. was aware of this issue at the time of the audit, and we confirmed it independently. As it was present in the reviewed commit, we've documented it here for completeness.

The `eth_call` implementation has a design flaw where the high-gas pool can be monopolized by a few requests with large gas limits. The high-gas pool is configured with only one worker thread (`fiber::PriorityPool high_gas_pool_{1, 2, true}`), making resources extremely limited.

The code logic shows the following:

```
use_high_gas_pool = (gas_specified
    && txn.gas_limit > MONAD_ETH_CALL_LOW_GAS_LIMIT)
```

When requests specify gas limits above the threshold, they enter the high-gas pool. Additionally, there is a fallback path from the low pool `REVERT` to the high pool, which is equally affected by this vulnerability.

The time-out mechanism only checks queuing time without limiting execution time:

```
if (std::chrono::steady_clock::now() - call_begin > timeout)
```

An attacker can send just one to two `eth_call` requests with massive `gas_limit` values to completely monopolize the high pool thread, causing all legitimate high-gas business requests to be rejected or time out.

Impact

This may lead to complete failure of high-gas pool functionality, preventing legitimate high-gas requests from being processed.

Recommendations

Implement proper execution time-out mechanisms for better resource management.

Remediation

TBD

3.8. The `operator()` type-conversion bug in `BytesHashCompare`

Target	category/core/bytes_hash_compare.hpp		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

In the `BytesHashCompare` template struct in core/bytes_hash_compare.hpp, the `operator()` function incorrectly returns a `bool` value by implicitly converting a `size_t` hash value:

```
template <class Bytes>
struct BytesHashCompare
{
    size_t hash(Bytes const &a) const
    {
        return komihash(a.bytes, sizeof(Bytes), 0);
    }

    bool equal(Bytes const &a, Bytes const &b) const
    {
        return memcmp(a.bytes, b.bytes, sizeof(Bytes)) == 0;
    }

    bool operator()(Bytes const &a) const
    {
        return hash(a);  // Error: size_t implicitly converted to bool
    }
};
```

The `hash()` function returns a `size_t` hash value, but the `operator()` implicitly converts this to `bool`. This means only the hash value `0` returns `false`, while all other hash values return `true`, completely breaking the intended hash-function functionality.

Impact

Implicit type conversion breaks hash-function semantics and could cause performance issues if used with STL containers that expect proper hash values.

Recommendations

Either remove the unused `operator()` function for safety or correct it to return the proper `size_t` hash value instead of converting to `bool`.

Remediation

This issue has been acknowledged by Category Labs, Inc., and a fix was implemented in commit `8a8b9bf2`.

3.9. Undefined behavior in `static_lru_cache` iterator usage

Target	category/core/lru/static_lru_cache.hpp		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The `insert` function in `static_lru_cache` contains undefined behavior by using an iterator after it has been invalidated by the `list_.erase()` operation:

```
void insert(Key const &key, Value const &value) noexcept
{
    auto it = map_.find(key);
    // Case 1: Key exists (update operation)
    if (it != map_.end()) {
        it->second->val = value;
        update_lru(it->second);
    }
    // Case 2: Key doesn't exist (insert operation)
    else {
        // Get iterator pointing to the last element in the list
        auto it = std::prev(list_.end());
        // Remove the entry from the hash table map_
        map_.erase(it->key);

        // Remove this node from the list
        list_.erase(it);

        // Undefined behavior: Using invalidated iterator
        it->key = key;
        it->val = value;

        list_.insert(list_.begin(), *it);
        map_[key] = it;
    }
}
```

After `list_.erase(it)` is called, the iterator `it` becomes invalid according to C++ standard, and any subsequent use of this iterator results in undefined behavior.

Impact

While this undefined behavior may not manifest as crashes in the current implementation due to the boost intrusive list's null disposer, it violates the C++ API contract and could lead to unpredictable behavior in different environments or compiler optimizations.

Recommendations

Restructure the code to avoid using the iterator after it has been invalidated by the erase operation.

Remediation

This issue has been acknowledged by Category Labs, Inc., and a fix was implemented in commit `0bac2c0b` ↗.

3.10. Dangling pointer in `cleanup_free` function

Target	category/core/cleanup.c		
Category	Coding Mistakes	Severity	Informational
Likelihood	N/A	Impact	Informational

Description

The `cleanup_free` function has an incorrect parameter declaration that prevents proper pointer nullification after memory deallocation:

```
void cleanup_free(char *const *const ptr)
{
    assert(ptr);
    if (*ptr) {
        free(*ptr);
    }
}
```

The current parameter type `char *const *const ptr` means that

1. `ptr` itself is a constant pointer and cannot point to other addresses, and
2. `*ptr` is also a constant pointer, and the address it points to cannot be modified through `*ptr`.

Because `*ptr` cannot be modified due to the `const` qualifier, the pointer still points to the freed memory block after the memory is freed, resulting in a dangling pointer. If the caller subsequently uses this pointer, a use-after-free vulnerability could occur.

Impact

This may lead to use-after-free vulnerabilities if callers attempt to use the pointer after `cleanup_free` is called, though the current impact is limited as this function appears to have minimal usage.

Recommendations

Change the parameter type from `char *const *const ptr` to `char **ptr` and add pointer nullification after freeing memory.

Remediation

TBD

3.11. Operator `int64_t` does not implement sign extension

Target	category/core/offset.hpp		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The `off48_t` class in offset.hpp is intended to store a 48-bit signed integer offset to save space (six bytes instead of eight):

```
constexpr operator int64_t() const
{
    std::array<char, 8> a{};
    if constexpr (std::endian::native == std::endian::little) {
        std::copy_n(&a_[0], 6, &a[0]);
    }
    else {
        std::copy_n(&a_[0], 6, &a[2]);
    }
    return std::bit_cast<int64_t>(a);
}
```

However, the current implementation only performs zero extension, not sign extension.

Impact

If the highest bit (bit 47) of a 48-bit value is 1 (indicating a negative number), the converted 64-bit value will be positive, resulting in a numerical error. Although the lack of sign extension can lead to hidden dangers, it is not used in the Monad code at the time of writing.

Recommendations

Consider supporting the sign extension or removing the current dead code.

Remediation

This issue has been acknowledged by Category Labs, Inc., and a fix was implemented in commit `e3ecd320` ↗.

3.12. Undefined behavior in bit_util.h

Target	category/core/bit_util.h		
Category	Coding Mistakes	Severity	Informational
Likelihood	N/A	Impact	Informational

Description

In the category/core/bit_util.h file, the `bit_div_floor` function is implemented as follows:

```
/**
 * finds the largest integer n such that n * 2^b <= x
 */
[[gnu::always_inline]] static inline unsigned long
bit_div_floor(unsigned long const x, unsigned long const b)
{
    return x >> b;
}
```

This function takes two unsigned long values, `x` and `b`, and finds the largest integer for `n` that satisfies the condition `n*2^b <= x`. However, when `x` is 10 and `b` is 64, the return value `n` is 10, which may indicate undefined behavior. We generally assume that `10>>64` should be 0.

Impact

When `x == 10`, `b == 64`, and `n == 10`, the commented condition `n * 2^b` is equal to `10*(2**64)`. This will yield 0 for unsigned long values, thus satisfying the condition `0 <= 10`. However, the return value `n == 10` is not the largest integer solution in this case; the largest integer solution should be the maximum value of an unsigned long. Undefined behavior in this case will result in incorrect calculations.

However, the probability of such undefined behavior is extremely low and such undefined behavior usually does not cause security risks.

Recommendations

Consider adding defensive checks to the code to catch any edge cases where this unexpected input happens.

Remediation

This issue has been acknowledged by Category Labs, Inc., and a fix was implemented in commit `edd82c2c` ↗.

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. Core infrastructure

The Core module provides foundational infrastructure services for the Monad system, including memory management, asynchronous I/O, fiber scheduling, cryptographic hashing, event logging, and synchronization primitives.

Architecture

The Core module provides fundamental infrastructure functionality for the entire Monad system. The memory-management system employs a layered design, containing the `HugeMem` allocator, `BatchMemPool` object pool, and EVM-specific ad hoc cache allocators. The asynchronous I/O system is based on the Linux `io_uring` implementation, providing batch-submission optimization through `Ring` class encapsulation.

Fiber scheduling uses Boost.Fiber to implement user-space cooperative scheduling. Moreover, `PriorityPool` manages multiple priority queues, implementing producer-consumer patterns through `buffered_channel`, and `PriorityAlgorithm` implements multilevel feedback queues, supporting work stealing and load balancing.

The hashing and cryptography module supports Keccak256 and Blake3 algorithms. Keccak256 includes both the C implementation and AVX2 assembly-optimized versions, while Blake3 implements parallel hashing and tree structures. All hash functions provide unified C++/C dual-language APIs.

The event-logging system implements a lock-free design based on shared memory. The ring buffer uses mmap and atomic operations, supporting multiple-producer single-consumer patterns. Event iterators provide safe traversal interfaces, using `pidfd_open` to monitor writer process status.

Synchronization primitives focus on ultra-low latency design — `SpinLock` is implemented using `atomic_flag`, employing intelligent backoff strategies and CPU pause instructions. The system includes built-in performance statistics to collect lock-contention information.

Attack surface

The Core module does not directly expose external attack surfaces, but as system infrastructure, the correctness and robustness of its internal functionality is critical. The module provides core services such as memory management, I/O processing, and task scheduling to upper-layer components, and any implementation defects could potentially be exploited by malicious input from upper-layer components.

Memory allocators need to handle various allocation requests from upper layers, including abnormally sized allocations and frequent allocation/deallocation patterns. The fiber-scheduling system needs to fairly handle tasks of different priorities, preventing task starvation or resource monopolization. The event-logging system needs to handle high-frequency event writes, ensuring data integrity and system stability.

4.2. Execution engine

The execution engine processes transactions and executes smart contracts, implementing both standard Ethereum execution logic and Monad-specific extensions, including parallel execution, state management, precompiled contracts, and the Staking system.

Architecture

The execution engine is divided into the standard Ethereum execution layer and Monad system extension layer. Block execution is coordinated through the `execute_block` function, using `fiber::PriorityPool` to manage parallel-transaction execution and fiber scheduling. Transaction execution is divided into two levels: `ExecuteTransactionNoValidation` focuses on basic execution logic, while `ExecuteTransaction` adds validation, state management, and receipt-generation functionality.

The EVMC host environment implements the standard EVMC interface through the `EvmcHostBase` base class and `EvmcHost` template. The system supports different EVM versions through template specialization, determining behavioral differences at compile time. The host environment is responsible for storage access, balance queries, contract calls, log recording, and account access control.

State management employs a three-layer architecture: the original layer preserves original state snapshots, the current layer uses `VersionStack` to manage modification versions, and the logs layer records all state changes. The `VersionStack` supports version creation and rollback for nested calls, ensuring complete state recovery when execution fails.

The precompiled contract system includes standard Ethereum precompiles (ecrecover, SHA-256, RIPEMD-160, identity, expmod, elliptic curve operations, BLS12-381 operations, etc.) and extended precompiles (e.g., `p256_verify`). The system supports conditional activation, controlling the availability of different precompiles through chain configuration.

The call-tracing system implements execution tracing through the `CallTracerBase` interface, supporting call-frame recording, self-destruct operation tracking, and complete execution-path recording. The system provides both `NoopCallTracer` and complete `CallTracer` implementations.

Monad system extensions include Staking precompiled contracts, providing complete validator management, delegation, undelegation, reward distribution, and system call functionality. The contracts use complex storage mapping structures to manage validator states, delegation relationships, and reward accumulators.

Attack surface

The execution engine exposes functionality to external parties through transaction execution and contract calls.

- **EVM instruction interface.** External parties can interact with `EvmcHost` through `CALL`, `DELEGATECALL`, `CREATE`, `CREATE2` and other instructions in transactions. The call method in `EvmcHost` handles all external-call requests and needs to perform gas-limit checks, stack-depth limits, balance validation, and `create_inside_delegated` flag checks. Attention should be paid to
    - whether the permission propagation mechanism of delegated calls has risks of bypassing access control,
    - whether the `CREATE` instruction's address calculation and conflict detection can effectively prevent malicious overwriting of existing contracts, and
    - whether deeply nested calls would cause excessive growth in `VersionStack` version history, leading to linear memory usage growth and CPU consumption issues in state rollback operations.
- **Precompiled contract interface.** External parties can submit input data to precompiled contracts for cryptographic operations. The system contains numerous precompiles for elliptic curve and cryptographic operations, including ecrecover for signature recovery, expmod for large integer operations, BLS12-381 for pairing checks, and so on. Attention should be paid to
    - whether input validation of these precompiled contracts is sufficient
    - whether they can prevent boundary-condition attacks and exceptional input handling, and
    - whether gas pricing accurately reflects computational complexity.

  Some precompiles have computational complexity that exhibits nonlinear relationships with input size, requiring verification of whether gas-pricing discrepancies exist that could lead to low-cost high-consumption attack risks.
- **State-management interface**: External parties can influence the state-management system through carefully crafted transaction sequences, with `VersionStack` handling version control and rollback operations for nested calls. State read-write dependencies and account-access patterns in parallel-transaction execution can all be influenced externally. Attention should be paid to
    - whether state-management version control has inconsistency risks,
    - whether `VersionStack` push and pop operations have strict error handling to prevent state leakage from version mismatches,
    - whether parallel execution has race conditions leading to nondeterministic results,
    - whether self-destruct operations in `EvmcHost` can ensure atomicity of balance transfers and contract deletion state transitions, and
    - whether compatibility across different EVM versions is guaranteed.
- **Staking contract interface**: External parties interact with the Staking system through

user operation interfaces such as `delegate`, `undelegate`, `withdraw`, and `claim_rewards`, while the system exposes system-call interfaces including `syscall_on_epoch_change`, `syscall_reward`, and `syscall_snapshot`. Complex storage mapping structures and linked-list traversal mechanisms provide rich interaction pathways for external parties. The `get_delegators_for_validator` and `get_validators_for_delegator` functions traverse potentially unbounded linked-list structures. Attention should be paid to

- whether these economic operations have overflow risks,
- whether linked-list traversal could lead to memory-usage explosion,
- whether the `RefCountedAccumulator` mechanism in reward calculations can ensure reference-count accuracy,
- whether system-call permission validation is strict enough to prevent unauthorized triggering, and
- whether it can effectively prevent ordinary transactions from forging system-level state modifications.

4.3. RPC interface

The RPC module exposes blockchain functionality to external clients through JSON-RPC interfaces, with the primary focus on `eth_call` execution using dual-tier fiber pools for load management and state-override capabilities for transaction simulation.

Architecture

The RPC module implements `eth_call` functionality through `monad_eth_call_executor`, employing a dual-tier fiber pool architecture for load separation. The low-gas pool (`low_gas_pool*`) uses multithread multifiber configuration to handle simple calls, while the high-gas pool (`high_gas_pool*`) uses single-thread dual-fiber configuration to handle complex calls. The system sets gas thresholds as pool-selection criteria and implements intelligent retry mechanisms to handle out-of-gas situations.

The state override system allows temporary modification of account balances, nonce values, contract code, storage states, and storage differences through the `monad_state_override` structure. Each `eth_call` executes in an independent State copy, accessing underlying state data through `TrieRODb`, ensuring complete isolation between simulated execution and main chain state.

Attack surface

The RPC module exposes execution engine functionality to external parties through the `eth_call` interface. External attackers can interact with the system through the following pathways: transaction-data fields in `eth_call` requests, block-header parameters, sender-address parameters, account balance, nonce values, contract code, storage state, and storage difference settings in state override configurations. Gas-limit parameters directly affect request-pool allocation and retry logic. The system also exposes call-tracing functionality switches and control parameters such as the `gas_specified` flag.

These input interfaces all require RLP decoding, parameter validation, and state construction processes, providing attackers with multiple potential attack entry points. If these attack surfaces are not adequately protected,

- malicious input could trigger parsing errors or buffer overflows in the RLP decoder, leading to service crashes;
- improper state override configurations could bypass validation logic to create inconsistent states;
- gas-limit parameter abuse could lead to incorrect resource-pool allocation and system performance degradation;
- massive malicious requests could fill queues and block legitimate processing; and
- deeply nested call chains could lead to memory exhaustion,

ultimately affecting the availability and response performance of the entire RPC service.

4.4. StateSync

The StateSync protocol enables distributed state synchronization between Monad nodes, implementing versioned protocol communication with client-server architecture for efficient Merkle-Patricia–trie data transmission and verification.

Architecture

The StateSync system implements distributed state synchronization, using versioned protocols to support compatibility between different protocol versions. The system includes both client and server implementations.

The client manages synchronization state through `monad_statesync_client_context`, maintaining progress tracking for 256 prefixes, with each prefix corresponding to a protocol instance. The client manages state caches, code hash sets, and block-header history records, using `deltas` and `buffered` two-tier caching to handle account and storage updates. The system implements a periodic commit mechanism, triggering state commits every million updates.

The server side implements efficient state data querying and transmission through trie traversal. The server uses the `TraverseMachine` interface to traverse the Merkle-Patricia trie, filtering and sending relevant state data based on prefixes. The system supports deletion-handling mechanisms, transmitting deleted account and storage states during interversion synchronization.

Protocol messages include `REQUEST` (synchronization request), `TARGET` (target setting), `DONE` (completion confirmation), and `UPSERT` series messages (`CODE`, `ACCOUNT`, `STORAGE`, `ACCOUNT_DELETE`, `STORAGE_DELETE`, `HEADER`). The `StatesyncProtocolV1` implements specific message-processing logic, containing `send_request` for sending synchronization requests and `handle_upsert` for handling state updates.

The network layer implements multitransport protocol support through function pointer abstraction, including `statesync_server_recv` for receiving data, `statesync_server_send_upsert` for sending state updates, and `statesync_server_send_done` for sending completion messages to callback interfaces.

Attack surface

The StateSync protocol's design assumes communication with trusted nodes, with the system lacking built-in node authentication and authorization mechanisms. The network layer implements transport protocols through function pointer abstraction but does not include peer node authentication or encrypted transmission protection. Protocol messages are processed directly, relying on security guarantees provided by the network layer or higher layers to ensure the trustworthiness of communication counterparts.

External parties can affect the system through the following interfaces: prefix, `prefix_bytes`, target, from, until, `old_target` parameters in protocol messages, account data, storage key-value pairs, contract code, block-header data in `UPSERT` messages, and raw byte streams at the network transport layer.

Although the system implements data-integrity verification mechanisms, including state-root verification, blockchain `parent_hash` chain checks, contract code Keccak256 hash verification, and so forth, these mechanisms primarily guard against data-transmission errors and partial malicious data injection and cannot completely defend against attacks from untrusted nodes.

Security concerns that require attention include

- whether untrusted nodes can cause out-of-bounds access to the protocol array through maliciously crafted prefix parameters,
- whether the RLP-decoding process of `UPSERT` messages has parsing errors and memory safety issues,
- whether malicious nodes providing incorrect `TARGET` messages could mislead synchronization to incorrect states,
- whether block-header verification mechanisms have bypass risks,

and so on.

5. Assessment Results

During our assessment on the scoped Monad targets, we discovered 12 findings. No critical issues were found. One finding was of high impact, five were of medium impact, one was of low impact, and the remaining findings 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.