Zellic

Prepared for
James Hunsaker
Category Labs, Inc.

Prepared by
Bryce Casaje
Jinseo Kim
Zellic

August 7, 2025

Monad RPC
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. 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 RPC'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:

- Are there any reliability issues?
- Does the codebase contain any undocumented functional differences from the API specification?
- Are there undesirable side effects or unintended state changes from RPC calls?
- Are there any unexpected panics or crashes from external input?
- Is the codebase resilient to logical Denial of Service (DoS) attacks (e.g. range-based attacks)?

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 RPC targets, we discovered six findings. No critical issues were found. Four findings were of medium impact, one was of low impact, and the remaining finding was informational in nature.

Breakdown of Finding Impacts

Impact Level	**Count**
🟥 Critical	0
🟧 High	0
🟨 Medium	4
🟩 Low	1
⬜ Informational	1

2. Introduction

2.1. About Monad RPC

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

> 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.
>
> The RPC component exposes a standard Ethereum RPC API.

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 RPC Targets

Type	Rust
Platform	Monad
Target	monad-bft
Repository	https://github.com/category-labs/monad-bft ↗
Version	`be342260a8875c6d0ada60857017ec093a04b844`
Programs	`monad-rpc` `monad-ethcall`

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 3 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

Jinseo Kim
Engineer
jinseo@zellic.io

2.5. Project Timeline

The key dates of the engagement are detailed below.

July 7, 2025	Kick-off call
August 11, 2025	Start of primary review period
August 25, 2025	End of primary review period

3. Detailed Findings

3.1. Debug namespace methods enabled by default without authentication

Target	monad-rpc		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	High	**Impact**	Medium

Description

All debug namespace methods are enabled by default in the RPC interface without requiring explicit configuration or authentication.

```
enabled_methods!(
    admin_ethCallStatistics,
    debug_getRawBlock,
    debug_getRawHeader,
    debug_getRawReceipts,
    debug_getRawTransaction,
    debug_traceBlockByHash,
    debug_traceBlockByNumber,
    debug_traceCall,
    debug_traceTransaction,
    // ...
);
```

The debug methods lack time-outs and resource limits, making them particularly dangerous. For example, the `debug_traceBlockByNumber` RPC method executes all transactions in the block specified by the block number with a tracer, and does not have a concurrency limit like the `rate_limiter` semaphore used in `eth_call`.

In comparison, Geth requires explicit enablement via the `--http.api=debug` command-line flag and typically disables debug methods in production environments.

Impact

These expensive tracing operations can consume excessive CPU and memory, making the node unresponsive and enabling denial-of-service attacks. Concurrent debug requests can overwhelm the node, causing resource exhaustion that affects legitimate RPC traffic.

Recommendations

Add a configuration flag to conditionally enable debug methods, disabled by default. In addition, implement time-outs and resource limits for all debug tracing operations.

Alternatively, add authentication/authorization mechanisms for administrative methods.

Remediation

This issue has been acknowledged by Category Labs, Inc., and a fix was implemented in PR #2332 ↗.

3.2. Race condition in state query methods allows stale-data retrieval

Target	monad-rpc		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

The methods `eth_getBalance`, `eth_getCode`, `eth_getStorageAt`, and `eth_getTransactionCount` retrieve data from state before validating that the state is still available and canonical. The `.await` between these operations creates a race-condition window where the blockchain state may change between the data retrieval and availability check.

```
let block_key = get_block_key_from_tag_or_hash(triedb_env,
    params.block_number).await?;
let account = triedb_env
    .get_account(block_key, params.account.0)  // (1) data is retrieved first
    .await
    .map_err(JsonRpcError::internal_error)?;

match triedb_env
    .get_state_availability(block_key)         // (2) state availability is
    checked after an await
    .await
    .map_err(JsonRpcError::internal_error)?
{
    true => Ok(format!("0x{:x}", account.balance)),
    false => Err(JsonRpcError::block_not_found()),
}
```

This pattern is repeated in all of the above methods, where the `.await` yields control back to the async runtime, allowing other tasks to potentially modify the underlying state before the availability check. During this window, blockchain reorgs or state pruning could make the retrieved data invalid while the availability check produces inconsistent results.

This race condition is dependent on the underlying `Triedb` implementation as it could be implemented in a safe way where `get_account()` and `get_state_availability()` always return consistent results for the same `block_key`. However, if they operate on different storage layers, use different caching mechanisms, or have different consistency guarantees, the same `block_key` could produce contradictory results between the two calls.

In contrast, methods like `eth_call` correctly check state availability first:

```
let block_key = get_block_key_from_tag_or_hash(triedb_env,
    params.block()).await?;
let version_exist = triedb_env
    .get_state_availability(block_key)
    .await
    .map_err(JsonRpcError::internal_error)?;
if !version_exist {
    return Err(JsonRpcError::block_not_found());
}

let mut header = match triedb_env
    .get_block_header(block_key)
    .await
    .map_err(JsonRpcError::internal_error)?
{
    Some(header) => header,
    // ...
```

Impact

Clients may receive stale or invalid account data that appears legitimate but represents outdated blockchain state.

The inconsistent behavior undermines data-integrity guarantees and can cause applications to make decisions based on unreliable state information, potentially leading to failures in dependent systems.

Recommendations

Adopt the same pattern used in `eth_call` for all state query methods.

Examine the underlying `Triedb` implementation to verify whether `get_account()` and `get_state_availability()` can actually return inconsistent results for the same `block_key`.

If the race condition proves to be real, consider implementing atomic operations such as `get_account_if_available()` methods that check availability and retrieve data in a single operation.

Add test cases that simulate concurrent access during pruning operations.

Remediation

TBD

3.3. Missing resource protection enables denial-of-service attacks

Target	monad-rpc		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	High	**Impact**	Medium

Description

Multiple RPC methods lack proper resource-protection mechanisms, allowing clients to submit oversized requests or trigger long-running operations that can exhaust server resources.

1. **Transaction size.** The `eth_sendRawTransaction` method does not validate the size of transaction data after decoding:

```
match TxEnvelope::decode(&mut &params.hex_tx.0[..]) {
    Ok(tx) => {
        // ...
    }
    Err(e) => Err(JsonRpcError::txn_decode_error())
}
```

2. **Filter complexity.** The `eth_getLogs` method does not validate the complexity of incoming filter parameters:

```
let MonadEthGetLogsParams { filters } = p;
let logs = chain_state
    .get_logs(
        filters,
        max_block_range,
        use_eth_get_logs_index,
        dry_run_get_logs_index,
        max_finalized_block_cache_len,
    )
    .await?;
```

3. **Execution time-outs.** The `eth_call` method lacks execution time-outs, allowing indefinite hanging:

```
match eth_call(
    tx_chain_id,
    txn,
    header.header,
    sender,
    block_number,
    block_round,
    eth_call_executor,
    &state_overrides,
    trace,
    gas_specified,
).await
```

While there is a global HTTP-request size limit configured via Actix's `PayloadConfig` (ranging from 8KB to 2MB in different configurations), there are no specific per-method validation limits.

Additionally, `eth_call` has gas-limit protection but no execution time-outs.

Impact

Missing resource protection creates multiple denial-of-service attack vectors through different forms of resource exhaustion.

Oversized transaction payloads can consume excessive memory during decoding, while complex filter queries require extensive database operations that can overwhelm the system. Large state override sets consume significant memory and CPU during temporary state tree construction. Long-running `eth_call` operations can tie up executor threads indefinitely, especially when calling contracts with infinite loops or computationally expensive operations.

These unbounded operations can make the RPC interface unresponsive to legitimate requests and cause performance degradation.

Recommendations

Add configurable input size limits including transaction size limits, filter-complexity validation, and state override limits.

Implement execution time-out protection for `eth_call` and other potentially long-running EVM methods with a configurable time-out parameter.

Make all limits configurable through command-line parameters to allow operators to tune based on their node's capacity and expected workload.

Remediation

Regarding the `eth_call` execution timeout, Category Labs, Inc. noted that some resource protection mechanisms were already implemented: execution requests have a timeout configured in `EthCallExecutor`, and there is a semaphore mechanism to limit concurrency on ETH calls. The `eth_call` execution model limitations were a known design consideration prior to the audit.

3.4. Parsing invalid hex string that contains + character does not fail

Target	monad-rpc		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Low	**Impact**	Medium

Description

In monad-rpc/src/hex.rs, there are the two public functions `decode` and `decode_quantity` that decode the given hex string into bytes or an unsigned 64-bit integer:

```rust
pub fn decode(s: &str) -> Result<Vec<u8>, DecodeHexError> {
    // (...)

    let Some(noprefix) = s.strip_prefix("0x") else {
        return Err(DecodeHexError::ParseErr);
    };

    // (...)

    decode_even_suffix(noprefix)
}

// Must be prefixed by 0x
// No leading zeros allowed
pub fn decode_quantity(s: &str) -> Result<u64, DecodeHexError> {
    // (...)

    let Some(noprefix) = s.strip_prefix("0x") else {
        return Err(DecodeHexError::ParseErr);
    };

    // (...)

    u64::from_str_radix(noprefix, 16).map_err(|_| DecodeHexError::ParseErr)
}

fn decode_even_suffix(s: &str) -> Result<Vec<u8>, DecodeHexError> {
    debug_assert!(s.len() & 1 == 0);
    debug_assert!(!s.is_empty());
    (0..s.len())
        .step_by(2)
```

```
          .map(|i| u8::from_str_radix(&s[i..i + 2],
      16).map_err(|_e| DecodeHexError::ParseErr))
          .collect()
  }
```

These functions utilize the `from_str_radix` function of the Rust standard library to decode hex characters into an integer. However, the `from_str_radix` function accepts the optional plus sign (+) before digits. It means `decode("0x12+3")` will return the result `0x1203` instead of an error.

Impact

RPC will incorrectly accept some invalid hex strings that contain the + sign before digits. This could be problematic if a user provides incorrect hex strings to RPC assuming that it would filter out such erroneous input.

Recommendations

Validate that a user-provided hex string does not contain the + sign before passing it to the `from_str_radix` function.

Remediation

This issue has been acknowledged by Category Labs, Inc., and a fix was implemented in PR #2233 ↗.

3.5. Missing fee-cap validation allows excessive transaction fees

Target	monad-rpc		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The `eth_sendRawTransaction` method does not validate transaction fees against a reasonable maximum cap, potentially allowing users to submit transactions with excessive fees.

```
match TxEnvelope::decode(&mut &params.hex_tx.0[..]) {
    Ok(tx) => {
        // ...
    }
}
```

Impact

The absence of fee-cap validation primarily affects users rather than the security of the node.

Users may accidentally submit transactions with excessive fees due to wallet bugs or misconfiguration, resulting in economic loss that could be significant.

Typos in fee specification could result in substantial financial loss. The lack of protection against fee errors degrades the overall user experience of interacting with the RPC interface.

Recommendations

Add a configurable fee-cap parameter such as `--rpc-tx-fee-cap` with a default of 1 ETH.

Validate both `gasPrice` and `maxFeePerGas` against the configured cap during transaction submission to catch excessive fees before processing.

Remediation

TBD

3.6. Base-fee validation uses hardcoded block zero instead of current chain tip

Target	monad-rpc		
Category	Business Logic	**Severity**	Informational
Likelihood	High	**Impact**	Informational

Description

The `base_fee_validation` function in src/handlers/eth/txn.rs uses a hardcoded `current_block = 0` instead of querying the actual current chain tip for EIP-1559 base-fee validation.

```
fn base_fee_validation(max_fee_per_gas: u128, base_fee: impl BaseFeePerGas) ->
    bool {
    let current_block = 0; // TODO: this can get latest block from triedb in
    future
    let block_threshold = 1000;

    let Some(min_potential_base_fee) =
        base_fee.min_potential_base_fee_in_range(current_block,
    block_threshold)
```

This function is called during transaction submission in `eth_sendRawTransaction` to validate that the transaction's `maxFeePerGas` is sufficient:

```
if !base_fee_validation(tx.max_fee_per_gas(), base_fee_per_gas) {
    return Err(JsonRpcError::custom(
        "maxFeePerGas too low to be include in upcoming blocks".to_string(),
    ));
}
```

Additionally, the underlying fee calculation uses a `FixedFee` struct that returns a static base fee, violating EIP-1559's dynamic fee-adjustment mechanism:

```
/// The base fee never changes
pub struct FixedFee {
    // ...
}

impl BaseFeePerGas for FixedFee {
    fn base_fee_at(&self, _block_num: u64) -> Option<u128> {
        Some(self.fixed_base_fee)  // Always returns same value
    }
}
```

Impact

The hardcoded `current_block = 0` and static base-fee implementation deviates from standard EIP-1559 behavior.

If this is intended to match Ethereum mainnet compatibility, this deviation could lead to a transaction acceptance inconsistency where the node behaves differently than other Ethereum clients.

Since this was the expected behavior at the time of the audit, the Severity and Impact are classified as Informational. Nevertheless, a correction is still recommended to ensure compatibility with the Ethereum mainnet.

Recommendations

Replace the hardcoded `current_block = 0` with a query to retrieve the actual latest block number from the chain state.

Implement proper EIP-1559 dynamic base-fee calculation that considers parent block data and network conditions, rather than returning fixed values.

Replace the `FixedFee` implementation with a dynamic fee provider that calculates base fees according to EIP-1559 specifications and chain history.

Remediation

Category Labs, Inc. noted that this was intentionally designed to be a static base fee at the code version in time for the audit.

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. monad-rpc

This section documents the architecture and threat model for monad-rpc.

Architecture

The RPC server runs on Tokio's multithreaded runtime with a configured worker thread count of 2 using the macro `#[tokio::main(flavor = "multi_thread", worker_threads = 2)]`. A global Rayon thread pool handles compute-intensive tasks with configurable thread counts specified by `args.compute_threadpool_size`.

The system uses Actix Web as the HTTP server framework, configured with two worker processes via `.workers(2)`. The server binds to configurable address and port combinations and establishes two primary endpoints: a `POST` route at `/` for JSON-RPC requests handled by `rpc_handler` and a WebSocket endpoint at `/ws/` for real-time communication.

Request size limits are enforced as a global HTTP body limit through `web::PayloadConfig::default().limit(args.max_request_size)`, rather than per-method validation, providing configurable payload bounds. The server includes middleware layers for tracing, timing, and optional metrics collection.

The monad-rpc server is comprised of several core components:

- **`EthTxPoolBridge`**, which manages communication with the validator's transaction pool via IPC, enabling transaction submission
- **`TriedbEnv`**, which provides access to blockchain state data with configurable concurrency, caching, and request buffering
- **`ChainState`**, which coordinates between `TriedbEnv` and `ArchiveReader` to provide unified access to blockchain data, with separate caches for finalized and voted blocks
- **`EthCallExecutor`**, which handles contract simulation requests in a dedicated environment with configurable thread and fiber counts, an LRU cache, and configurable queuing time-outs
- **`ArchiveReader`**, which accesses historical data from AWS S3 or MongoDB backends, with configurable fallback mechanisms

JSON-RPC method dispatch occurs through a centralized `rpc_handler` function that processes incoming requests and routes them to appropriate handler implementations based on method names. The system includes optional RPC comparison functionality for validating responses against reference implementations.

A semaphore-based system controls concurrent access to expensive operations like `eth_call` and `debug_traceCall`. The system implements configurable limits for batch requests, response sizes, and gas consumption across different operation types.

The application uses a comprehensive CLI-based configuration system that controls endpoint addresses, feature flags, and operational parameters for all major system components.

Attack surface

External attackers can interact with the system through two primary channels:

1. **JSON-RPC HTTP endpoint.** A `POST` endpoint at `/` accepts JSON-RPC requests, which are parsed by the `rpc_handler` and dispatched via the `enabled_methods!` macro in src/handlers/mod.rs. The size of these requests is limited by `web::PayloadConfig` and the number of batch requests is controlled by the `batch_request_limit`.
2. **WebSocket endpoint.** A `/ws/` endpoint allows for streaming input and real-time communication.

The accepted JSON-RPC methods, defined in the `enabled_methods!` macro, include a wide range of handlers. The `debug_*` and `trace_*` methods are enabled by default. The `admin_ethCallStatistics` method is also included but is gated by the `enable_admin_eth_call_statistics` runtime flag.

External input flows through the HTTP server, is parsed, and is then routed to the appropriate handler. These handlers then interact with the backend components (`EthTxPoolBridge`, `TriedbEnv`, `ChainState`, `EthCallExecutor`, `ArchiveReader`) to fulfill the request.

The system employs multiple layers of input validation through JSON-RPC parsing and method-specific parameter validation. Transaction processing handles complex encoding schemes and transaction formats, while state access operations must handle invalid references to blockchain entities without compromising system stability.

Denial-of-service risks

The multithreaded architecture distributes work across Tokio runtime workers, compute thread pools, and specialized execution environments. Resource exhaustion in any execution context can impact system availability.

Request size limits and batch-request constraints provide basic protection against large payload attacks. Range-based operations allow clients to specify potentially expensive query parameters that may require extensive database traversal.

State-integrity considerations

The RPC interface includes both read-only query operations and state-modifying transaction-submission capabilities. Transaction-submission methods are explicitly designed to modify system state through the validator's transaction-processing pipeline.

Query operations including state lookups, block retrieval, and historical data access are intended to be read-only and should not produce unintended side effects on system state or validator operations.

Contract simulation methods execute bytecode in isolated environments that should not affect persistent blockchain state. These operations may temporarily modify execution contexts but must maintain proper isolation from the live validator state.

Administrative and debugging methods provide system introspection capabilities that expose internal operational data without modifying core blockchain state.

Protocol compliance and compatibility

The RPC interface implements standard JSON-RPC protocols with extensions specific to blockchain operations. The system aims to maintain compatibility with established Ethereum RPC conventions while supporting Monad-specific functionality.

5. Assessment Results

During our assessment on the scoped Monad RPC targets, we discovered six findings. No critical issues were found. Four findings were of medium impact, one was of low impact, and the remaining finding was 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.