Zellic

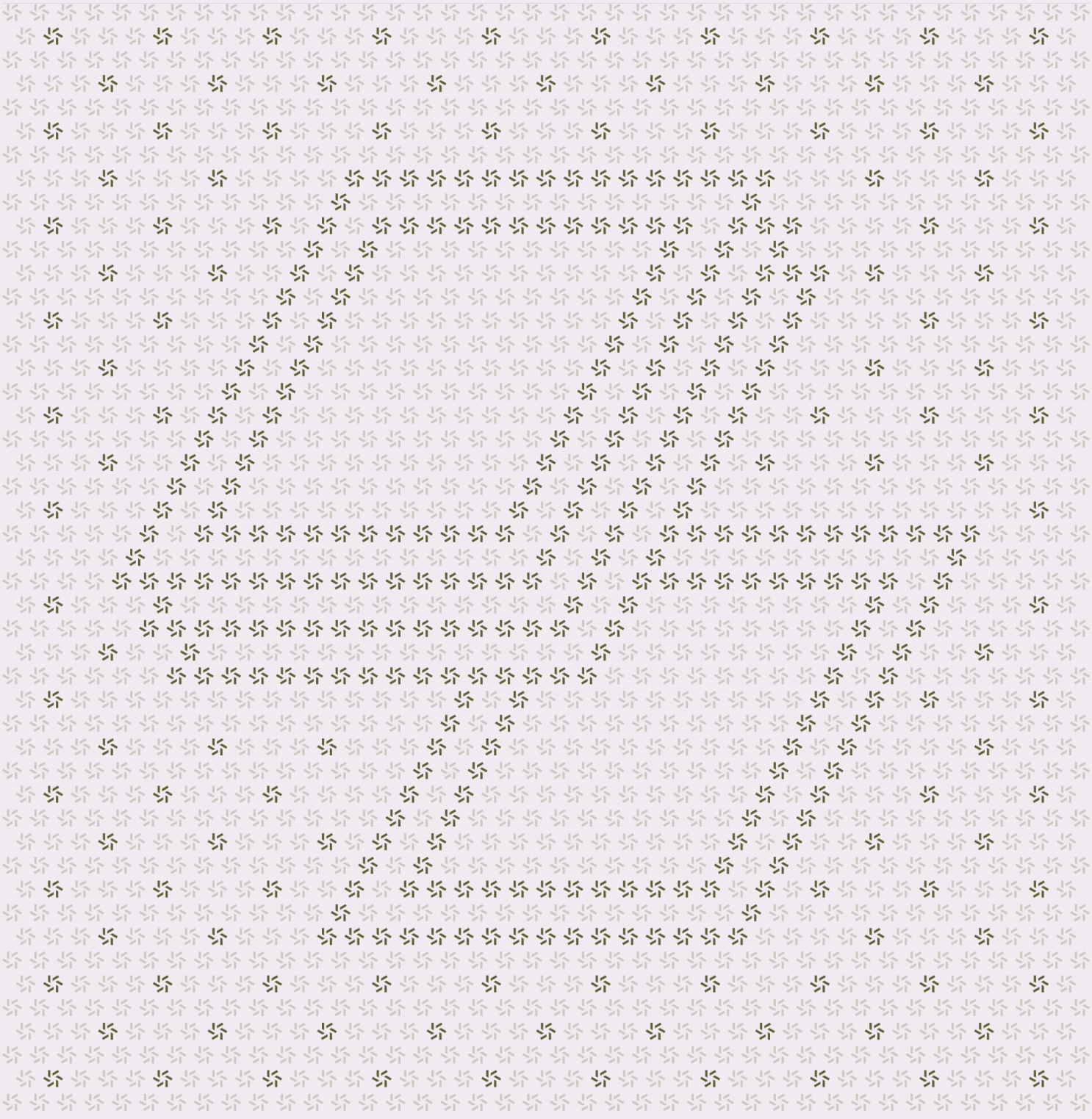
Prepared for
Jordan Oroshiba
Astria

Prepared by
Jisub Kim
Jaeeu Kim
Zellic

January 27, 2025

Astria Shared Sequencer Oracle

Blockchain 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 Astria from January 8th to January 16th, 2025. During this engagement, Zellic reviewed Astria Shared Sequencer Oracle'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 the `market_map` and `oracle` components ported correctly and functioning as expected?
- Have critical changes been made that could halt the chain or disrupt operations?
- Is the price feed secure against exploitation by Byzantine validators?

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

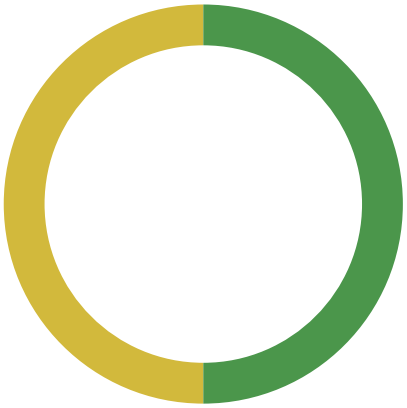
Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

1.4. Results

During our assessment on the scoped Astria Shared Sequencer Oracle modules, we discovered two findings. No critical issues were found. One finding was of medium impact and one was of low impact.

Breakdown of Finding Impacts

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



2. Introduction

2.1. About Astria Shared Sequencer Oracle

Astria is a Shared Sequencer Network with Celestia underneath. It provides ordering guarantees with soft commitments to chains, relying on Celestia's DAS network to provide firm commitments and broad data availability. Astria is developing a decentralized sequencing layer that can be shared among multiple rollups. The Astria Stack sequences arbitrary data for rollups, makes it accessible to rollup nodes, enables easy retrieval and verification, and batches rollup blocks before posting them to Celestia.

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 modules.

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:

Astria Shared Sequencer Oracle Modules

Type	Rust
Platform	Rust-compatible
Target	Astria changes between ffbd00...0a03d8, PR#1878, and PR#1900
Repository	https://github.com/astriaorg/astria ↗
Version	`0a03d83bb2eec88f837fe9f512fae6c71decd38c`

Programs	`crates/*`
	`Excluding the following:`
	`- astria-sequencer/src/connect/market_map/state_ext.rs`
	`- astria-sequencer/src/connect/market_map/storage/*`
	`- astria-sequencer/src/connect/oracle/state_ext.rs`
	`- astria-sequencer/src/connect/oracle/storage/*`

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of two person-weeks. The assessment was conducted by two consultants over the course of 2 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

The following consultants were engaged to conduct the assessment:

Jisub Kim
Engineer
jisub@zellic.io

Jaeeu Kim
Engineer
jaeeu@zellic.io

2.5. Project Timeline

The key dates of the engagement are detailed below.

January 8, 2025	Start of primary review period
January 9, 2025	Kick-off call
January 16, 2025	End of primary review period

3. Detailed Findings

3.1. The `num_currency_pairs` is not incremented

Target	astria-sequencer/src/connect/oracle/state_ext.rs		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

The `put_price_for_currency_pair` function is responsible for either updating the price of an existing `CurrencyPair` or creating a new `CurrencyPairState` if a currency pair does not exist. However, when a new `CurrencyPairState` is created, the `num_currency_pairs` state, which keeps track of the total number of currency pairs, is not incremented.

```
#[instrument(skip_all)]
async fn put_price_for_currency_pair(
    &mut self,
    currency_pair: CurrencyPair,
    price: QuotePrice,
) -> Result<()> {
    let state = if let Some(mut state) = self
        .get_currency_pair_state(&currency_pair)
        .await
        .wrap_err("failed to get currency pair state")?
    {
        // price update logic ..
        // [..]
    } else {
        let id = self
            .get_next_currency_pair_id()
            .await
            .wrap_err("failed to read next currency pair ID")?;
        let next_id = id.increment().wrap_err("increment ID overflowed")?;
        self.put_next_currency_pair_id(next_id)
            .wrap_err("failed to put next currency pair ID")?;
        CurrencyPairState {
            price: Some(price),
            nonce: CurrencyPairNonce::new(0),
            id,
        }
        // no put_num_currency_pairs here
    };
    self.put_currency_pair_state(currency_pair, state)
```

```
        .wrap_err("failed to put currency pair state")
}
```

Impact

This mismatch could lead to issues in parts of the application that depend on an accurate `num_currency_pairs` count. It may cause skipped entries in reports and incorrect indexing when iterating over currency pairs.

Recommendations

Consider updating the `put_price_for_currency_pair` function to increment the `num_currency_pairs` state when a new `CurrencyPairState` is created.

Remediation

This issue has been acknowledged by Astria, and a fix was implemented in commit 7057387e ↗.

3.2. Inaccurate representation of negative prices

Target	astria-core/src/connect/types.rs		
Category	Business Logic	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The Astria sequencer treats the price as an unsigned integer:

```
// astria-core/src/sequencerblock/v1/block.rs
pub struct Price {
    currency_pair: CurrencyPair,
    price: crate::connect::types::v2::Price,
    decimals: u64,
}

// astria-core/src/connect/types.rs
pub struct Price(u128);
```

The price of real-world equities is sometimes negative. In the event this happens, when cast to an unsigned quantity, the protocol will instead assume it is very large, causing unintended behavior of the oracle.

Impact

This behavior might cause unintended actions within the oracle, such as an incorrect representation of market conditions or failure to update prices accurately.

Recommendations

Consider supporting signed integers for price representation to handle negative values correctly.

Remediation

This issue has been acknowledged by Astria, and a fix was implemented in PR 1991 ↗

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. Astria Oracle

Description

The Astria Oracle is responsible for providing the protocol with the current price feeds. It is a critical component as the protocol relies on the Oracle to determine the value of the assets.

The validator nodes are responsible for providing the Oracle with the current price feeds. The validators have sidecars that are responsible for fetching the price of feeds from external sources and submitting them, voting on these prices until consensus. If two thirds of the validators agree on the voting of oracle, the price is considered valid, and it will be stored in the state.

Components

The Astria Oracle have two components:

OracleComponent Oracle stores and manages currency-pair data, including mappings from pairs to IDs, states, and price quotes.

MarketMapComponent MarketMap stores and manages data about multiple markets, including their tickers, decimal precision, provider configurations, and metadata.

Process

The Oracle component goes through the following process:

1. `init_chain` sets `vote_extensions_enable_height` (this is the height where the voting extension is enabled) and sets up MarketMapComponent and OracleComponent.
2. `prepare_proposal` makes `extended_commit_info` from the last commit and attaches it to the proposal.
3. `process_proposal` validates the `extended_commit_info` in the proposal.
4. `finalize_block` calls `apply_prices_from_vote_extensions` to write the price of pairs to the state.

Vote extensions

Vote extensions in the network allow validators to fetch off-chain data and include it in their votes, which are then broadcasted and committed on-chain. This process is managed by `vote extension` handlers. Since vote extensions are only available to other validators at the next block height, oracle data is always one height behind. Additionally, each validator maintains a local view of vote extensions, meaning there is no single canonical set. To ensure determinism, the network relies on the next proposer's local view as the authoritative version.

The oracle data can only be committed to the chain if more than 2/3 of the validator power submits valid vote extensions. If fewer than 2/3 of validators submit valid vote extensions, the chain does not halt. In this case, blocks will be finalized but no new oracle data won't be published for that block.

Handler

- `extend_vote` retrieves price information from the sidecar oracle client and transforms it into a vote extension. It returns an empty vote extension if price retrieval fails.
- `transform_oracle_service_prices` transforms oracle service prices by converting currency pairs to their corresponding IDs and filtering out unknown pairs.
- `verify_vote_extension` validates oracle vote extension by checking price count and individual price lengths.

ProposalHandler

- `prepare_proposal` prepares a proposal by validating and pruning vote extensions from the previous block.
- `validate_proposal` validates a proposed block's `extended_commit_info` against the last commit and state rules.
- `get_id_to_currency_pair` retrieves a mapping of currency-pair IDs to their corresponding currency-pair information from the state.
- `validate_id_to_currency_pair_mapping` validates that the currency-pair mapping is consistent with expected values, checking for missing, extra, or mismatched pairs.
- `validate_vote_extensions` validates vote extensions by checking for repeated voters, signature validity, voting power, and other consensus-related constraints. Two thirds of validators must agree on the price for it to be considered valid.
- `validate_extended_commit_against_last_commit` compares the current extended commit info with the last commit to ensure consistency in round, vote, and validator information.
- `apply_prices_from_vote_extensions` applies prices from vote extensions to the state, storing quote prices for each currency pair.

Oracle-price calculation

The current system uses a median-based approach to calculate prices from validators. Instead of relying on weighted pricing, where some validators may have more influence, the median simply takes the middle value from a sorted list. This makes it naturally resistant to extreme outliers. Even if a validator is compromised and proposes an incorrect price, it will not significantly affect the result

as long as the majority of validators provide reasonable values.

For example, if validators report 3.1, 3.2, 3.2, 3.1, 3,000, and 3, the median would still be 3.2 despite the outlier (3,000). As more validators participate, the effect of a single malicious actor becomes even smaller. While the median approach is not perfect — it could still be manipulated if enough validators collude — it is a straightforward method to ensure reliable pricing in most situations. Adding extra checks, like outlier detection, could make it even stronger.

Oracle client timeout and retry logic

The `new_oracle_client` function uses an exponential back-off retry mechanism during the initial connection attempt to the oracle sidecar. The retry duration starts at 100 ms and doubles with each attempt until a maximum delay of 10 seconds is reached. The total retry time sums to approximately five minutes. Although this maximum delay is significant compared to the two-second block-confirmation time, it only affects the initial start-up process. If the connection fails after all retries, the sequencer exits with an error.

```
/// Returns a new Connect oracle client or `Ok(None)` if `config.no_oracle` is
    true.
///
/// If `config.no_oracle` is false, returns `Ok(Some(...))` as soon as a
    successful response is
/// received from the oracle sidecar, or returns `Err` after a fixed number of
    failed re-attempts
/// (roughly equivalent to 5 minutes total).
#[instrument(skip_all, err)]
async fn new_oracle_client(config: &Config) ->
    Result<Option<OracleClient<Channel>>> {
    if config.no_oracle {
        return Ok(None);
    }
    let uri: Uri = config
        .oracle_grpc_addr
        .parse()
        .context("failed parsing oracle grpc address as Uri")?;
!    let endpoint = Endpoint::from(uri.clone()).timeout(Duration::from_millis(
!        config.oracle_client_timeout_milliseconds,
!    ));

    let retry_config =
    tryhard::RetryFutureConfig::new(MAX_RETRIES_TO_CONNECT_TO_ORACLE_SIDECAR)
!        .exponential_backoff(Duration::from_millis(100))
!        .max_delay(Duration::from_secs(10))
        .on_retry(
            |attempt, next_delay: Option<Duration>, error: &eyre::Report| {
                let wait_duration = next_delay
```

```
                    .map(humantime::format_duration)
                    .map(tracing::field::display);
                warn!(
                    error = error.as_ref() as &dyn std::error::Error,
                    attempt,
                    wait_duration,
                    "failed to query oracle sidecar; retrying after backoff",
                );
                async {}
            },
        );
```

In normal operations, the oracle client applies a configurable timeout (`ASTRIA_SEQUENCER_ORACLE_CLIENT_TIMEOUT_MILLISECONDS`, default is 1,000 ms) to each RPC request. If a request times out, the sequencer does not retry; instead, it provides an empty vote extension to ensure the network remains operational. This design prioritizes liveness over data accuracy during transient oracle-connectivity issues.

```
impl Handler {
    pub(crate) fn new(oracle_client: Option<OracleClient<Channel>>) -> Self {
        Self {
            oracle_client,
        }
    }

    pub(crate) async fn extend_vote<S: StateReadExt>(
        &mut self,
        state: &S,
    ) -> Result<abci::response::ExtendVote> {
        let Some(oracle_client) = self.oracle_client.as_mut() else {
            // we allow validators to *not* use the oracle sidecar currently,
            // so this will get converted to an empty vote extension when
    bubbled up.
            //
            // however, if >1/3 of validators are not using the oracle, the
    prices will not update.
            bail!("oracle client not set")
        };

!       // if we fail to get prices within the timeout duration, we will
    return an empty vote
!       // extension to ensure liveness.
!       let rsp = match oracle_client.prices(QueryPricesRequest {}).await {
!           Ok(rsp) => rsp.into_inner(),
!           Err(e) => {
!               bail!("failed to get prices from oracle sidecar: {e:#}",);
```

```
!             }
!         };
```

Attack surface

Malicious validators and external sources can provide false prices to the Oracle.

5. Assessment Results

At the time of our assessment, the reviewed code was not deployed to the Mainnet.

During our assessment on the scoped Astria Shared Sequencer Oracle modules, we discovered two findings. No critical issues were found. One finding was of medium impact and one was of low impact.

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.