Zellic

Empiric Oracle

Smart Contract Security Assessment

October 25, 2022

Prepared for:

Jonas Nelle and Robert Kelly

Empiric Network

Prepared by:

Jasraj Bedi and Sampriti Panda

Zellic Inc.

Contents

About Zellic

Zellic was founded in 2020 by a team of blockchain specialists with more than a decade of combined industry experience. We are leading experts in smart contracts and Web3 development, cryptography, web security, and reverse engineering. Before Zellic, we founded perfect blue, the top competitive hacking team in the world. Since then, our team has won countless cybersecurity contests and blockchain security events.

Zellic aims to treat clients on a case-by-case basis and to consider their individual, unique concerns and business needs. Our goal is to see the long-term success of our partners rather than simply provide a list of present security issues. Similarly, we strive to adapt to our partners' timelines and to be as available as possible. To keep up with our latest endeavors and research, check out our website zellic.io or follow @zellic_io on Twitter. If you are interested in partnering with Zellic, please email us at hello@zellic.io or contact us on Telegram at https://t.me/zellic_io.

1 Executive Summary

Zellic conducted an audit for Empiric Network from August 29th to September 16th, 2022.

Our overall assessment of the project is that it was well-written and demonstrated a strong familiarity with Starknet, was well-structured, and often employed defensive implementation and designs that suggest resilience to security issues during development. Despite the restrictions that development with Starknet and Cairo impose, Empiric Network has done an excellent job not only implementing non-trivial features but doing so in a way that is safe in the relatively novel Starknet ecosystem.

We applaud Empiric Network for their attention to detail, as evident by Empiric Network self-identifying issues early on in the course of the audit.

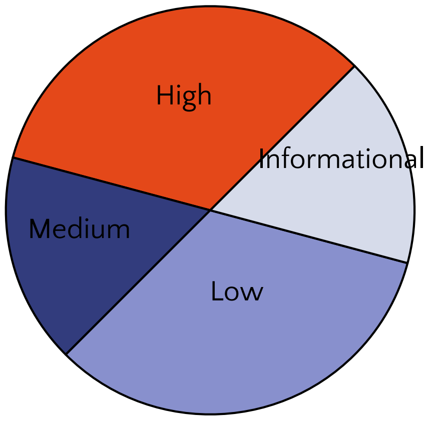
Zellic thoroughly reviewed the Empiric Oracle codebase to find protocol-breaking bugs as defined by the documentation and to find any technical issues outlined in the Methodology section (2.2) of this document.

During our assessment on the scoped Empiric Oracle contracts and non-contract code, six findings were discovered. Two high issues, one medium issue, and two low issues were identified, and the remaining finding was informational in nature due to updates in Starknet's platform.

Additionally, Zellic identified and recorded suggestions during the course of the audit for Empiric Network's benefit in the Discussion section (4) at the end of the document.

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	2
Medium	1
Low	2
Informational	1



2 Introduction

2.1 About Empiric Oracle

Empiric Oracle is a decentralized, transparent and composable oracle network, leveraging state-of-the-art zero-knowledge cryptography. Empiric Oracle partners with market makers and exchanges who sign and timestamp their own high quality, robust data and send it directly on-chain.

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 open-source 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. We analyze the scoped smart contract code using automated tools to quickly sieve out and catch these shallow bugs. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so forth as necessary. We also perform a cursory review of the code to familiarize ourselves with the contracts.

Business logic errors. Business logic is the heart of any smart contract application. We manually review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents. We also thoroughly examine the specifications and designs themselves for inconsistencies, flaws, and vulnerabilities. This involves use cases that open the opportunity for abuse, such as flawed tokenomics or share pricing, arbitrage opportunities, and so forth.

Complex integration risks. Several high-profile exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. We perform a meticulous review of all of the contract's possible external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so forth.

Code maturity. We review for possible improvements in the codebase in general. We look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradeability weaknesses, centralization risks, and so forth.

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; we assign it on a case-by-case basis based on our professional judgment and experience. As one would expect, both the severity and likelihood of an issue affect its impact; for instance, a highly severe issue's impact may be attenuated by a very low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Similarly, Zellic organizes its reports such that the most important findings come first in the document rather than being 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 numerous soft factors, such as our clients' threat models, their business needs, their project timelines, and so forth. We aim to provide useful and actionable advice to our partners that consider their long-term goals rather than simply provide a list of security issues at present.

2.3 Scope

The engagement involved a review of the following targets:

Empiric Oracle Contracts

Repository	https://github.com/42labs/Empiric
Versions	`8924f00901f1a4ff796473e8381028b9e5c4c281`
Programs	• Oracle • Proxy • Publisher Registry • Empiric-Package (Python)
Type	Cairo, Python
Platform	Starknet

2.4 Project Overview

Zellic was contracted to perform a security assessment with two consultants for a total of six person-weeks. The assessment was conducted over the course of three calendar weeks.

Contact Information

The following project managers were associated with the engagement:

Jasraj Bedi, Co-founder
jazzy@zellic.io

Stephen Tong, Co-founder
stephen@zellic.io

The following consultants were engaged to conduct the assessment:

Jasraj Bedi, Engineer
jazzy@zellic.io

Sampriti Panda, Engineer
sampriti@zellic.io

2.5 Project Timeline

The key dates of the engagement are detailed below.

August 29, 2022	Kick-off call
August 29, 2022	Start of primary review period
September 16, 2022	End of primary review period
TBD	Closing call

3 Detailed Findings

3.1 Key used in oracle entry storage is forgable by trusted publisher

- **Target**: oracle/library.cairo
- **Category**: Coding Mistakes
- **Likelihood**: Low
- **Severity**: High
- **Impact**: High

Description

The `oracle/library.cairo` code is responsible for much of the core implementation of the oracle itself. The oracle uses "entries" to record the current value for a given asset pair or other kinds of tracked elements. The oracle code defines a "publish entry" external function that allows callers to submit an entry to be recorded.

The main authorization check is done by checking that the caller's address is equal to the expected publisher address. The expected publisher address is reported by the publisher registry contract. This check ensures that this transaction can only be performed by a preconfigured publisher. While this check ensures that the caller is, indeed, a preconfigured publisher, it does not key the entry by this caller address.

Entries define multiple relevant properties. Namely, entries define a timestamp, the value, a pair id, a source, and a publisher.

```
struct Entry:
    member pair_id : felt
    member value : felt
    member timestamp : felt
    member source : felt
    member publisher : felt
end
```

The pair id represents a string of the pair of assets this entry tracks. For example, this could be the felt value that represents the string `"eth/usd"`. The other interesting property is the source. The source and the publisher are not necessarily the same. The publisher attests to the value of data from a particular source. Therefore, an entry submitted by a publisher could contain *any* source string desired.

Entries are stored in a map called `Oracle_entry_storage`, which is keyed by two values:

the entry's pair id and the entry's source. Because entry sources can be any value decided by the publisher and entries are not keyed by their publisher, rogue publishers can overwrite the values set by other publishers.

Impact

Approved publishers that have turned rogue can set entries for arbitrary sources and key ids even if those sources are the responsibility of other publishers.

Recommendations

Considering either keying on publisher address or tracking which sources a particular publisher is allowed to publish. This will require an additional check that the specified source is allowed to be published by the calling publisher.

Remediation

The issue was addressed in a later update.

3.2 Publish entry does not validate caller address is not `0`

- **Target**: oracle/library.cairo
- **Category**: Coding Mistakes
- **Likelihood**: N/A
- **Severity**: Informational
- **Impact**: Informational

Description

When contracts in Starknet are directly invoked, the `get_caller_address` function can return `0`. This is a relatively common error or default pattern in Starknet and Cairo, but it can cause security issues when this behavior is unexpected, like in the case of `get_caller_address`.

In the publish entry code of the `oracle/library.cairo` file, the caller address is checked against the publisher address. The publisher address is retrieved by calling into the publisher registry contract and fetching the address of the publisher with a given felt-converted string name. If the publisher specified by this string does not exist, the publisher registry will actually return `0` instead of throwing an error.

```
func Publisher_get_publisher_address{
    syscall_ptr : felt*, pedersen_ptr : HashBuiltin*, range_check_ptr
}(publisher : felt) -> (publisher_address : felt):
    let (publisher_address) = Publisher_publisher_address_storage.read(
    publisher)
    return (publisher_address)
end
```

This is because a read with a key that does not exist will return `0` values instead of throwing an error. Because no check is performed in the publisher registry that validates that non-zero values for publisher addresses will be returned, this allows the oracle code to check a `0` publisher address against a potentially `0` caller address, which can occur if the contract is invoked directly with `--no_wallet`.

As of Starknet 0.10.0 this will not be an issue, but it is recommended to validate that the publisher registry get publisher address method does not return `0` values and/or the oracle validates the caller is not `0`.

Impact

If allowed, for example—a pre-0.10.0 Starknet environment would allow a caller to impersonate a publisher as long as the publisher does not exist in the publisher registry. In combination with a previous finding, this would allow an attacker to publish arbitrary entries even if they were not previously added to the registry.

Recommendations

Validate, in either the publisher registry, that the returned publisher address is non-zero or that the caller address is not zero.

Remediation

The issue was addressed in a later update.

3.3 Mathematical expressions could produce incorrect values

- **Target**: oracle/library.cairo
- **Category**: Coding Mistakes
- **Likelihood**: Low
- **Severity**: Medium
- **Impact**: High

Description

It was observed in the yield curve cairo code that in `calculate_future_spot_yield_point` some multiplication occurs with numbers that have not been given an upper bound. While integer overflow conditions are not strictly limited to multiplication, this is where we're most likely to find valid conditions for overflow behavior.

In `calculate_future_spot_yield_point`, a call is made to `starkware.cairo.common.pow` where the exponent is `output_decimals + spot_decimals - future_decimals`. Based on how this function is called, `future_decimals` can, at least, be `1`. No reasonable upper bound exists for the exponent and `pow`, internally, performs unchecked multiplication. This means that the following expressions

```
# Shift future/spot to the left by output_decimals + spot_decimals -
    future_decimals
let (ratio_multiplier) = pow(10, output_decimals + spot_decimals -
    future_decimals)
let (shifted_ratio, _) = unsigned_div_rem(
    future_entry.value * ratio_multiplier, spot_entry.value
)
```

can result in integer overflow when performing the `pow` operation as the exponent cap is 2ˆ251. Note that this is not 251, but 2 raised to the 251. This will easily overflow the `ratio_multiplier`, causing the ratio to be an unexpected value.

Impact

Mathematical expressions can miscalculate, causing incorrect spot pricing.

Recommendations

Assert that the exponent passed to `pow` is less than some amount. Additional, provide additional assertions around entry valuation to ensure the provided number is reasonable and not at the limits of what a felt can support.

Remediation

The issue was addressed in a later update.

3.4 Faulty implementation of comparison function

- **Target**: lib/time_series/utils.cairo
- **Category**: Coding Mistakes
- **Likelihood**: Low
- **Severity**: Medium
- **Impact**: Medium

Description

The `are_equal` function in `time_series.utils` incorrectly assumes that the `is_nn` function checks if the argument is negative. According to the documentation, however, this function checks if the argument is non-negative. This leads to an incorrect implementation, causing the `are_equal` function to return bogus values.

```
func are_equal{range_check_ptr}(num1: felt, num2: felt) -> (_are_equal:
    Bool) {
    alloc_locals;
    let is_neg1 = is_nn(num1 - num2);
    let is_neg2 = is_nn(num2 - num1);
    let _are_equal = is_nn(is_neg1 + is_neg2 - 1);
    return (_are_equal,);
}
```

As an example, the `are_equal` function will have the following trace when run with arguments `(3, 4)`, wrongly returning that the numbers are equal:

```
is_neg1 = is_nn(3 - 4) => is_nn(-1) => 0;
is_neg2 = is_nn(4 - 3) => is_nn(1) => 1
_are_equal = is_nn(0 + 1 - 1) => is_nn(0) => 1
```

Impact

The faulty `are_equal` function is used as a helper function by other statistical calculation functions under time_series/, which could lead to incorrect results.

Recommendations

Rewrite the code according to the correct specification of the is_nn function: It returns 1 when the argument is non-negative.

Write more unit tests for individual library functions to catch any incorrect implementations and edge cases that might not show up in an integration test.

Remediation

The issue was addressed in a later update.

3.5 Incorrect use of library comparison function

- **Target**: compute*engines/rebase*denomination/RebaseDenomination.cairo
- **Category**: Coding Mistakes
- **Severity**: Low
- **Likelihood**: Low
- **Impact**: Low

Description

The `_decimal_div` function uses the `is_le` function to compare the number of decimals between the numerator and the denominator. The specification of the `is_le` function states that it returns 1 when the first argument is less than or equal to the second argument:

```
// Returns 1 if a ≤ b (or more precisely 0 ≤ b - a < RANGE_CHECK_BOUND).
// Returns 0 otherwise.
@known_ap_change
func is_le{range_check_ptr}(a, b) -> felt {
    return is_nn(b - a);
}
```

The implementation of `_decimal_div` assumes otherwise. The `is_le` function will return `TRUE` if `b_decimals ≤ a_decimals` or `a_decimals ≥ b_decimals`. This is different from the code below, which assumes that the two numbers can only be equal in the else branch.

```
let b_fewer_dec = is_le(b_decimals, a_decimals);
if (b_fewer_dec == TRUE) {
    // x > y
    a_to_shift = a_value;
    result_decimals = a_decimals;
    tempvar range_check_ptr = range_check_ptr;
} else {
    // x ≤ y
```

As a result, the case when the two numbers are equal is handled by the first if branch instead of the else branch as expected by the code.

Impact

The correctness of the `_decimal_div` function is not affected by the incorrect usage of the `is_le` function as the code for handling the first if branch and the equality leads to the same outcome.

However, this same mistake may show up in other places, and such assumptions should be carefully verified before using them in code.

Recommendations

Rearrange the if conditions so that the case of equality is handled by the if branch rather than the else branch.

Remediation

The issue was addressed in a later update.

3.6 Inconsistency in checking of stale entries

- **Target**: oracle/library.cairo
- **Category**: Coding Mistakes
- **Likelihood**: Low
- **Severity**: Low
- **Impact**: Low

Description

There are two places where entries are checked to be recent: `publish_entry` and `build_entries_array`.

The `publish_entry` verifies the following condition for new entries: `(current_timestamp - TIMESTAMP_BUFFER) ≤ new_entry.timestamp`

```
let (current_timestamp) = get_block_timestamp();
with_attr error_message("Oracle: New entry timestamp is too far in the
    past") {
    assert_le(current_timestamp - TIMESTAMP_BUFFER, new_entry.timestamp);
}
```

The `build_entries_array` checks the following condition to filter entries that are too old: `entry.timestamp ≤ current_timestamp - TIMESTAMP_BUFFER)`

```
let is_entry_stale = is_le(entry.timestamp, current_timestamp -
    TIMESTAMP_BUFFER);
let should_skip_entry = is_not_zero(is_entry_stale +
    not_is_entry_initialized);
```

Ideally both the checks should have the same statement; however, when we rearrange and list them, we see that there is a certain timestamp where the `publish_entry` states that the entry is fresh but `build_entries_array` says that the entry is stale.

```
// Entries are fresh if:
current_timestamp - TIMESTAMP_BUFFER <= new_entry.timestamp
current_timestamp - TIMESTAMP_BUFFER < entry.timestamp
```

Impact

If an entry is on the boundary of being stale, and it is published and fetched at the same timestamp, it will be rejected. While this is not a security concern, it is important to ensure that assumptions and invariants across the project are consistent with each other to prevent bugs from occuring in the future.

Recommendations

Ensure that both of the conditions are consistent with each other and check the same thing.

Remediation

The issue was addressed in a later update.

4 Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment.

4.1 Assert `admin_address_storage` is set to non-zero value in `only_admin`

The `only_admin` function, found in the `admin/library.cairo` file, is used to check if the caller is the specified admin for the contract. This code is used in a couple of contracts in the repository, allowing the protocol operators to perform maintenance tasks like adding publishers.

Currently, the `only_admin` function does not validate that the `admin_address_storage` variable was previously initialized. In an uninitialized state, the value of this variable is `0`. This, coupled with the fact that `get_caller_address` will return `0` if the contract is directly invoked, suggests that if improper initialization were to occur with the contract that callers that directly invoked this contract would be treated as administrators.

Upon review of the codebase, we found no such case of improper initialization of the admin code in any of the in-scope contracts. While this design decision did not rise to the level of a security issue as it was used correctly everywhere, it is our recommendation that the check that this field has been initialized be added for future security of any additional admin functionality.

It is worth noting, however, that similar to the "publish entry non-zero check" finding, that due to the transition to Starknet 0.10.0, this will likely not be an issue going forward.

4.2 Root and publisher key scheme

As part of the audit, we were consulted for ideas on how to address potential design and operational issues. One such topic of discussion was how to design against publisher key compromise. In this situation, a publisher may mismanage their publishing key, allowing an attacker or insider to access the key and begin publishing fraudulent entries to the protocol.

After some discussion and deliberation, we came up with a proposal to have publishers register two keys—a root key and a publisher key. The publisher key would actually allow for publishing entries to the oracle. The root key would only be able to set a given publisher's publishing key. The idea would be that publishers would keep their root key in cold storage, locked away. This would keep the key very safe and away from accidental leakage. Publishers would then use their publisher key on live systems to actually publish entries.

If this key ever need be rotated, the publisher can do that entirely themselves by using their root key to configure a new publisher key with the oracle. Optionally, an additional mechanism to rotate the root key with itself could be envisioned so that publishers can be particularly careful with the root key if they fear it has been compromised as well.

4.3 Pause functionality

For additional operational control of the contract, it may be advisable to build in a rapid protocol-pausing functionality if anything were to happen. For example, if a security issue were later identified, being able to pause the protocol before further harm was done, this would be ideal from an incident response standpoint. Even without the presence of a security issue in the protocol itself, it would help to pause the protocol or individual publishers if a publisher's key is determined to be compromised.

This does come with centralization risk, though, so that should be weighed against the pros and cons for this protocol. MPC could be an alternative to retain some decentralized properties despite the presence of the pause functionality.

4.4 Accuracy of data returned by Rebase Denomination

The `get_rebased_value_via_usd` function returns two values that help the user understand how the data was sourced: `last_updated_timestamp` and `num_sources_aggregated`. These are calculated using the maximum number of sources of each of the currency pairs and the latest of the timestamps when each of the pairs were updated.

This is a bit misleading as it does not signal to the user the risk of a stale or centralized oracle for one of the pairs. We recommend using the minimum function to calculate these values to better convey the accuracy and recency of the data to the users.

5 Audit Results

At the time of our audit, the code was not deployed to mainnet.

During our audit, we discovered six findings. Of these, two were high, one medium, two low, and one finding was a suggestion (informational). Empiric Network acknowledged all findings and implemented fixes.

5.1 Disclaimers

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 additional code added to the assessed project after the audit version of 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 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.

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.