Zellic

Pyth Governance

Smart Contract Security Assessment

May 16, 2022

Prepared for:

Jonathan Claudius

Pyth Data Association

Prepared by:

Filippo Cremonese and Jasraj Bedi

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 Introduction

1.1 About Pyth Governance

Pyth is a first party financial oracle with real-time market data on-chain. It aims to bring valuable financial market data to DeFi applications and the general public. Pyth intends to use the Solana smart contracts under audit as a tool to give stakeholders (holders of PYTH token) a way to make and vote proposals affecting Pyth's governance.

1.2 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, etc. 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, etc.

Complex integration risks. Several high-profile exploits have been the result of not any bug within the contract itself, but rather an unintended consequence of its 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, etc.

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

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 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, project timelines, etc. We aim to provide useful and actionable advice to our partners that consider their long-term goals, rather than simply a list of security issues at present.

1.3 Scope

The engagement involved a review of the following targets:

Pyth Governance contract

Repository	https://github.com/pyth-network/pyth-governance
Versions	`750a2e91bd64bb4132427413b337ce1023bb755c`
Programs	• Pyth Governance
Type	Rust
Platform	Solana

1.4 Project Overview

Zellic was contracted to perform a security assessment with two consultants, for a total of 2 person-week. The assessment was conducted over the course of 1 calendar week.

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:

Filippo Cremonese, Engineer
fcremo@zellic.io

Jasraj Bedi, Co-Founder
jazzy@zellic.io

1.5 Project Timeline

The key dates of the engagement are detailed below.

May 9, 2022	Start of primary review period
May 10, 2022	Follow-up call
May 13, 2022	End of primary review period

1.6 Disclaimer

This assessment does not provide any warranties on finding all possible issues within its scope; i.e., the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees on any additional code added to the assessed project after our assessment has concluded. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program. Finally, this assessment report should not be considered financial or investment advice.

2 Executive Summary

Zellic conducted an audit for Pyth Data Association from May 2nd to May 6th, 2022 on the scoped contracts and discovered 1 finding. We found the code to be correct, and did not discover exploitable security issues.

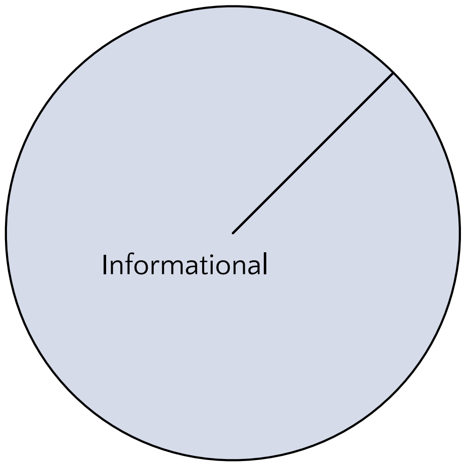
The discovered finding, while potentially very dangerous, is unexploitable and is thus reported as informational. Additionally, Zellic recorded its notes and observations from the audit for Pyth Data Association's benefit at the end of the document.

Zellic thoroughly reviewed the Pyth Governance codebase to find protocol-breaking bugs as defined by the documentation, or any technical issues outlined in the Methodology section of this document. Specifically, taking into account Pyth's threat model, we focused heavily on issues that would allow an attacker an unfair weight in Pyth governance voting, and issues that would allow withdrawal of funds that would bypass the exposure risk calculation as defined by the documentation.

Our general overview of the code is that it was correct and well structured. The code was not always intuitive, and we do think that there is some margin for improving code clarity, both by refactoring some the most complex functions and by adding more comments and documentation.

Breakdown of Finding Impacts

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



3 Detailed Findings

3.1 Missing account reload after cross program invocation

- **Target**: `withdraw_stake`
- **Category**: Coding Mistakes
- **Likelihood**: NA
- **Severity**: Low
- **Impact**: Informational

Description

Accounts that can be modified by cross program invocations (CPI) must be explicitly deserialized again after the CPI is performed. `withdraw_stake` invokes the token program to transfer tokens from the custody account to the user withdrawal account. After the CPI, the `amount` from the custody account is passed to `validate` without refreshing the information read from the account. The amount passed to `validate` is therefore the amount before the tokens are transferred.

```
transfer(
    CpiContext::from(&*ctx.accounts).with_signer(&[&[
        AUTHORITY_SEED.as_bytes(),
        ctx.accounts.stake_account_positions.key().as_ref(),
        &[stake_account_metadata.authority_bump],
    ]]),
    amount,
)?;

if utils::risk::validate(
    stake_account_positions,
    stake_account_custody.amount,
    unvested_balance,
    current_epoch,
    config.unlocking_duration,
)
```

Impact

This mistake makes the call to `validate` occurring after the transfer effectively useless. Fortunately the issue is not exploitable, as the function performs a call to `validate` immediately before the token transfer, using a computed amount corresponding to the amount after the transfer.

Recommendation

Currently, the call to `validate` after `transfer` does not have any effect; it cannot cause a transaction to revert, since the condition for a revert would have already been caught by the call to `validate` before the transfer. Therefore removing the call to `validate` after `transfer` is a valid remediation.

Alternatively, we recommend calling inserting a call to `.reload()` immediately after the call to `transfer`.

Remediation

TBD: Pending client feedback.

4 Discussion

The purpose of this section is to document miscellaneous observations the we made during the assessment. Those observations do not have a direct security impact, and are intended to increase code quality, robustness and/or efficiency.

4.1 Inefficient `get_unused_index` implementation

The `PositionData::get_unused_index` method iterates up to 100 times looking for an available `PositionData` slot in the `positions` array. This is because closed positions are marked by taking the specified `PositionData` and setting that entry to `None`.

If, instead, this was implemented by tracking the next available index (*we'll refer to this as `next_index`*) used to fill a position and swapping the last position with the newly closed position, there would never be gaps in the positions. This means that `get_unused_index` can just return `next_index` after ensuring that `next_index` is not out of bounds (less than `100`).

An example updated `PositionData` that would use this fast implementation for `get_unused_index` could look like the following:

```rust
pub struct PositionData {
    pub owner:     Pubkey,
    pub positions: [Option<Position>; MAX_POSITIONS],
    next_index: usize
}

impl PositionData {
    /// Finds first index available for a new position
    pub fn get_unused_index(&self) -> Result<usize> {
        if self.next_index < MAX_POSITIONS {
            Ok(next_index)
        } else {
            Err(error!(ErrorCode::TooManyPositions))
        }
    }

    pub fn add_position(&mut self, position: Position) -> Result<()> {
        let index = self.get_unused_index()?;
        self.positions[index] = Some(position);
        self.next_index += 1;

        Ok(())
    }

    pub fn delete_position(&mut self, position_index: usize) -> Result<()
    > {
        if position_index >= MAX_POSITIONS {
            return Err(error!(ErrorCode::PositionNotInUse));
        }

        match self.positions[position_index] {
            Some(_) => {
                self.next_index -= 1;
                self.positions[position_index] = self.positions[self.
    next_index];
                self.positions[self.next_index] = None;

                Ok(())
            },
            None => Err(error!(ErrorCode::PositionNotInUse))
        }
    }
}
```

4.2 Simplify `get_current_position` bounds checking

The logic of the current implementation of `Position::get_current_position` is non-trivial to understand from the perspective of an external developer or auditor. We find that it could be simplified by writing some of the results of the arithmetic operations and conditions to appropriately named variables, and then test bounds using those variables.

An example is provided:

```
match self.unlocking_start {
    None => Ok(PositionState::LOCKED),
    Some(unlocking_start) => {
        let unlock_finished = unlocking_start + (unlocking_duration as
    u64);

        let is_activated = self.activation_epoch ≤ current_epoch;
        let is_unlock_started = unlocking_start ≤ current_epoch;
        let is_unlock_finished = unlock_finished ≤ current_epoch;

        if is_activated && !is_unlock_started {
            Ok(PositionState::PREUNLOCKING)
        } else if is_unlock_started && !is_unlock_finished {
            Ok(PositionState::UNLOCKING)
        } else {
            Ok(PositionState::UNLOCKED)
        }
    }
}
```

4.3 Avoid using default case in `validate` match statement

The `validate` function is used to determine the level of risk a particular set of positions present. As part of this, `total_exposure`, `max_target_exposure`, and `governance_exposure` parameters are determined by iterating over the collection of targets and their exposures.

As it stands, targets can only be of type `VOTING` or `STAKING`, however if future target types were added in the future, the default pattern used in this `match` statement would automatically assume a strategy for assessing risk for the new kind of target.

Given the critical nature of `validate`, we suggest to avoid using the default `_ ⇒ ...` pattern and instead explicitly match against all the possibilities in the enum variant. If future target types were added, this would result in a compiler error rather than a silently accepted default exposure calculation.

An example is provided:

```
for (target, exposure) in &current_exposures {
    match target {
        Target::VOTING => { /* voting exposure calc */ }
        Target::STAKING { .. } => { /* staking exposure calc */ }
    }
}
```