Zeta Markets

Security Assessment

May 13th, 2024 — Prepared by OtterSec

Tamta Topuria tamta@osec.io

Nicola Vella nick0ve@osec.io

Robert Chen notdeghost@osec.io

Table of Contents

Appendices

01 — Executive Summary

Overview

Zeta Markets engaged OtterSec to assess the `zeta-staking` program. This assessment was conducted between May 1st and May 8th, 2024. For more information on our auditing methodology, refer to Appendix B.

Key Findings

We produced 3 findings throughout this audit engagement.

In particular, we identified a critical vulnerability where the stake start epoch is set to the current epoch when adding a stake, allowing users to exploit staking rewards by staking tokens for a fraction of an epoch to receive a full epoch's worth of rewards (OS-ZTM-ADV-00). Additionally, we pointed out that updating the name of a staking account could sometimes result in an unintended name being set (OS-ZTM-ADV-01).

We also recommended modifying the codebase to enhance efficiency and ensure adherence to best coding practices (OS-ZTM-SUG-00).

Scope

The source code was delivered to us in a Git repository at https://github.com/zetamarkets/zeta-staking. This audit was performed against commit c03fe59.

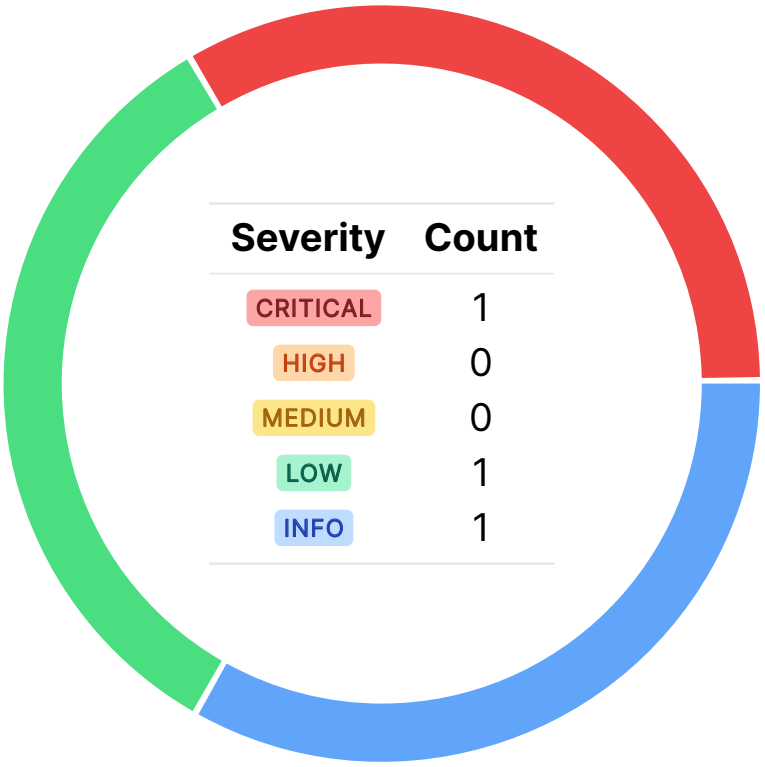
A brief description of the programs is as follows:

Name	Description
zeta-staking	The Zeta staking contracts provide the infrastructure for users to stake their Zeta tokens, manage their stake accounts, and claim rewards.

02 — Findings

Overall, we reported 3 findings.

We split the findings into **vulnerabilities** and **general findings**. Vulnerabilities have an immediate impact and should be remediated as soon as possible. General findings do not have an immediate impact but will aid in mitigating future vulnerabilities.



Severity	Count
CRITICAL	1
HIGH	0
MEDIUM	0
LOW	1
INFO	1

03 — Vulnerabilities

Here, we present a technical analysis of the vulnerabilities we identified during our audit. These vulnerabilities have *immediate* security implications, and we recommend remediation as soon as possible.

Rating criteria can be found in Appendix A.

ID	Severity	Status	Description
OS-ZTM-ADV-00	CRITICAL	RESOLVED	`stake_start_epoch` is set to the current epoch when adding a stake, allowing users to exploit staking rewards by staking tokens for a fraction of an epoch to receive a full epoch's rewards.
OS-ZTM-ADV-01	LOW	RESOLVED	The `StakeAccount`'s `name` is not cleared before updating, resulting in a wrong, unintended value being set as the `name` field.

Incorrect Stake Start Epoch Setting CRITICAL OS-ZTM-ADV-00

Description

`add_stake` incorrectly sets `stake_start_epoch` when adding stake to a vesting account. It sets the `stake_start_epoch` to the maximum value between the current and existing stake start epoch. As a result, users may start accruing rewards prematurely, potentially resulting in inconsistencies in reward distribution, especially if the protocol has a short `min_stake_duration_epochs`, as it may result in users receiving rewards earlier than expected.

>_ *state/stake_account.rs* rust

```rust
pub fn add_stake(&mut self, protocol_state: &ProtocolState, stake_to_add: u64) -> Result<()> {
    [...]
    match self.stake_state {
        StakeState::Vesting {
            stake_start_epoch,
            last_claim_ts,
        } => {
            self.stake_state = StakeState::Vesting {
                stake_start_epoch: protocol_state.epoch().max(stake_start_epoch),
                last_claim_ts,
            }
        }
        _ => {}
    }
    Ok(())
}
```

Proof of Concept

It is possible to stake tokens for a very small fraction of an epoch and still receive a whole epoch's worth of staking rewards, as shown below, where the current epoch is 14:

1. The attacker initializes a stake with a very short `stake_duration_epochs` (one epoch) and a small non-zero amount (one token). This stake is toggled immediately, setting the `stake_start_epoch` to the next epoch (epoch 15).
2. In epoch 15, the attacker attempts to claim rewards. However, since they are still in the `stake_start_epoch`, no rewards are claimed.
3. Towards the end of epoch 15, the attacker adds a large amount of stake (100,000 tokens) via `add_stake`. This resets the `stake_start_epoch` to epoch 15 again.

4. As soon as epoch 15 ends and epoch 16 begins, the attacker claims rewards. Because the `stake_start_epoch` is set to epoch 15 again due to the `add_stake` operation, they may claim rewards for the entire 100,001 tokens, despite only staking one token for a fraction of epoch 15.

Remediation

`add_stake` should set the `stake_start_epoch` to the next epoch (`get_next_epoch`) instead of the current epoch.

Patch

Resolved in 03a2a0d

Incorrect StakeAccount Name Setting LOW OS-ZTM-ADV-01

Description

In the `write_name` function, the `name` array is not cleared before a new value is set. Therefore, if a user calls the `edit_stake_account_name` instruction with a `name` value that is shorter than the previous one, the new string will be a concatenation of the new value and the remaining bytes of the previous value, which is not the user's intended outcome.

>_ *state/stake_account.rs* rust

```rust
pub fn write_name(&mut self, name: String) -> Result<()> {
    let nb = name.as_bytes();
    if nb.len() > MAX_ACC_NAME_LENGTH {
        return err!(ErrorCode::StakeAccountNameTooLong);
    }
    self.name[..nb.len()].copy_from_slice(nb);
    Ok(())
}
```

Remediation

The `write_name` function should clear the `name` array before writing a new value into it.

Patch

Resolved in 907a5e8

04 — General Findings

Here, we present a discussion of general findings during our audit. While these findings do not present an immediate security impact, they represent anti-patterns and may result in security issues in the future.

ID	Description
OS-ZTM-SUG-00	Recommendations for modifying the code base to enhance its efficiency and ensure adherence to best coding practices.

Code Optimization

OS-ZTM-SUG-00

Description

1. Currently `std::mem::size_of` is used to determine the size of accounts. When determining the size of an account structure, it is important to consider the actual serialized size rather than the size as reported by `std::mem::size_of`. For instance, in `StakeAccount`, the `stake_state` enumeration is 13 bytes in anchor. However, `mem::size_of` returns 16 bytes, making `StakeAccount` unnecessarily big. Thus, it would be more appropriate to utilize Borsh Deserialize's size.
2. The `bit_to_use` parameter may only be up to 64 bits. It is advisable to check explicitly that the value falls within this range rather than relying solely on overflow checks to catch potential issues.

Remediation

Implement the above-mentioned suggestions.

Patch

Resolved in 03a2a0d

A — Vulnerability Rating Scale

We rated our findings according to the following scale. Vulnerabilities have immediate security implications. Informational findings may be found in the General Findings.

CRITICAL

Vulnerabilities that immediately result in a loss of user funds with minimal preconditions.

Examples:

- Misconfigured authority or access control validation.
- Improperly designed economic incentives leading to loss of funds.

HIGH

Vulnerabilities that may result in a loss of user funds but are potentially difficult to exploit.

Examples:

- Loss of funds requiring specific victim interactions.
- Exploitation involving high capital requirement with respect to payout.

MEDIUM

Vulnerabilities that may result in denial of service scenarios or degraded usability.

Examples:

- Computational limit exhaustion through malicious input.
- Forced exceptions in the normal user flow.

LOW

Low probability vulnerabilities, which are still exploitable but require extenuating circumstances or undue risk.

Examples:

- Oracle manipulation with large capital requirements and multiple transactions.

INFO

Best practices to mitigate future security risks. These are classified as general findings.

Examples:

- Explicit assertion of critical internal invariants.
- Improved input validation.

B — Procedure

As part of our standard auditing procedure, we split our analysis into two main sections: design and implementation.

When auditing the design of a program, we aim to ensure that the overall economic architecture is sound in the context of an on-chain program. In other words, there is no way to steal funds or deny service, ignoring any chain-specific quirks. This usually requires a deep understanding of the program's internal interactions, potential game theory implications, and general on-chain execution primitives.

One example of a design vulnerability would be an on-chain oracle that could be manipulated by flash loans or large deposits. Such a design would generally be unsound regardless of which chain the oracle is deployed on.

On the other hand, auditing the program's implementation requires a deep understanding of the chain's execution model. While this varies from chain to chain, some common implementation vulnerabilities include reentrancy, account ownership issues, arithmetic overflows, and rounding bugs.

As a general rule of thumb, implementation vulnerabilities tend to be more "checklist" style. In contrast, design vulnerabilities require a strong understanding of the underlying system and the various interactions: both with the user and cross-program.

As we approach any new target, we strive to comprehensively understand the program first. In our audits, we always approach targets with a team of auditors. This allows us to share thoughts and collaborate, picking up on details that the other missed.

While sometimes the line between design and implementation can be blurry, we hope this gives some insight into our auditing procedure and thought process.