Veridise

Auditing Report

Hardening Blockchain Security with Formal Methods

FOR

RISC ZERO

ZKC Staking

Veridise Inc.

September 5, 2025

► **Prepared For:**

RISC Zero
https://risczero.com
https://beboundless.xyz

► **Prepared By:**

Benjamin Sepanski
Victor Faltings

► **Contact Us:**

contact@veridise.com

► **Version History:**

Sep. 5, 2025	V2 - Incorporate fixes to issues
Sep. 3, 2025	V1
Sep. 2, 2025	Initial Draft

Contents

1 Executive Summary

From Aug. 21, 2025 to Aug. 29, 2025, RISC Zero engaged Veridise to conduct a security assessment of their ZKC. The security assessment covered their ZKC token, staking logic which allows users to stake ZKC into veZKC for voting rights and rewards, and their logic for minting rewards based on proof-of-verifiable work. Veridise conducted the assessment over 2 person-weeks, with 2 security analysts reviewing the project over 1 week on commits `f6f8f89b` (ZKC) and `8bd70a89` (boundless). The review strategy involved a tool-assisted analysis of the program source code performed by Veridise security analysts as well as thorough code review.

Project Summary. The security assessment covered a RISC Zero zkVM program and four core contracts: `ZKC`, `veZKC`, `StakingRewards`, and `PovwMint`. `ZKC` is an ERC-20 fungible token. This token is granted to initial recipients by two designated admins, then subsequently minted through staking or proof-of-verifiable work (Povw) rewards. The token supply over time is fixed on-chain, with an initial supply of one billion `ZKC` tokens, followed by regular emissions every two days, called an "epoch". The epoch emission rate is set so that the supply increases by a fixed percentage each year, starting at 7% in the first year and slowly decreasing to an annual rate of 3% from year eight onward.

Users can gain the emitted `ZKC` as rewards through two mechanisms. The first is by staking. `ZKC`-holders may stake their `ZKC` tokens into `veZKC`. The `veZKC` contract follows the ERC-721 standard, tracking each user's stake as a non-fungible token. Users' total staked `ZKC` is used to determine their voting power and reward power. Users may delegate their stake to contribute to other accounts' voting or reward power rather than their own. All staking and delegation is checkpointed in both per-user and global histories, allowing votes and reward computations to be computed reliably for past epochs.

Users may initiate an unstake at any time. From this point on, no one may add stake to the users' position, and the action is checkpointed so that their stake no longer contributes to either per-user or global voting or reward power. After thirty days elapse, the user may withdraw their staked `ZKC`, destroying their position. Note that a user with a pending unstake may still have voting/reward power delegated *to* them, and so still have voting or reward power in a given epoch.

Staking rewards are claimed through the `StakingRewards` contract. A user may claim rewards from any *past* epoch. Staking rewards are computed based on the users' share of the total reward power at the end of the epoch. Twenty-five percent of the emissions from that epoch are allocated to staking rewards. For example, if 1000 `ZKC` were emitted in epoch 100, and at the end of that epoch Alice held 50% of the staked `veZKC` tokens, she could claim 125 `ZKC` tokens as rewards in that epoch.

Povw rewards are claimed through the `PovwMint` contract. Users create a zero-knowledge proof using the RISC Zero zkVM. This proof searches for `WorkLogUpdate` events emitted from the out-of-scope `PovwAccounting` contract. Each `WorkLogUpdate` corresponds to a verifiable amount of computation performed to create ZK proofs. In each epoch, users may claim rewards based

on the proportion of verifiable work they posted during that epoch. The `PovwMint` contract verifies the off-chain proof, checks its inputs against contract state, and then mints `ZKC` tokens to the rewards recipients. The remaining seventy-five percent of per-epoch token emissions are set aside for Povw-rewards. To prevent other protocols from leveraging this yield to their own advantage, Povw-rewards are capped based on the amount of `ZKC` that was staked by the recipient during the rewards epoch.

Code Assessment. The ZKC developers provided the source code of the ZKC contracts for the code review. The source code appears to be original code written by the ZKC developers. It contains some documentation in the form of READMEs and documentation comments on functions and storage variables. To facilitate the Veridise security analysts' understanding of the code, the ZKC developers also provided RFCs describing the intended behavior in detail.

The source code contained a test suite, which the Veridise security analysts noted had high coverage and tested for negative cases such as claiming in multiple epochs. Several files in the source code also indicate that the developers use linting and fuzzing.

Summary of Issues Detected. The security assessment uncovered 9 issues, 1 of which was assessed to be of high or critical severity by the Veridise analysts. In particular, V-ZKC-VUL-001 describes how using multiple reward recipients allows a user to bypass the reward cap. The Veridise analysts also identified 1 low-severity issue in which duplicate events are emitted, see V-ZKC-VUL-002. Finally, the Veridise analysts reported 6 warnings, and 1 informational finding.

The ZKC developers fixed all the issues except for a single warning issue, V-ZKC-VUL-008, which was acknowledged but deemed to be an acceptable risk. This issue describes a mechanism by which one user may slow down another user's ability to claim rewards. However, the scenario is expensive and will ultimately terminate with all of the user's rewards distributed to the user. All other fixes to issues were validated by the Veridise analysts.

Recommendations. After conducting the assessment of the protocol, the security analysts had a few suggestions to secure ZKC.

Atomic Deployment. As mentioned in V-ZKC-INFO-001 and V-ZKC-VUL-004, non-atomic initialization may be taken advantage of by attackers. Currently, a version one of `ZKC` is already deployed. The upgrade should atomically call the `initializeV2` function along with deploying the new contracts.

Use Proxy. Currently, `PovwMint` has no way to update the image ID. If the RISC Zero zkVM is updated, this could prevent the protocol from switching to the latest version. When the Veridise team discussed this with the RISC Zero developers, they indicated their intent was to deploy the `PovwMint` behind a proxy.

Additional Underflow Checks. As delegation may increase or decrease power, signed values are used. While Veridise analysts believe all workflows prevent under/over-flow, additional checks to enforce this property would provide defense-in-depth and prevent similar attacks in future versions.

Disclaimer. We hope that this report is informative but provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the system is secure in all dimensions. In no event shall Veridise or any of its employees be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the results reported here.

2 Project Dashboard

Table 2.1: Application Summary.

Name	Version	Type	Platform
ZKC	`f6f8f89b`	Solidity	Ethereum
boundless	`8bd70a89`	Solidity, Rust	Ethereum, RISC Zero

Table 2.2: Engagement Summary.

Dates	Method	Consultants Engaged	Level of Effort
Aug. 21–Aug. 29, 2025	Manual & Tools	2	2 person-weeks

Table 2.3: Vulnerability Summary.

Name	Number	Acknowledged	Fixed
Critical-Severity Issues	0	0	0
High-Severity Issues	1	1	1
Medium-Severity Issues	0	0	0
Low-Severity Issues	1	1	1
Warning-Severity Issues	6	6	5
Informational-Severity Issues	1	1	1
TOTAL	9	9	8

Table 2.4: Category Breakdown.

Name	Number
Maintainability	2
Data Validation	2
Logic Error	1
Incorrect/Missing Events	1
Access Control	1
Rounding Error	1
Frontrunning	1

3 Security Assessment Goals and Scope

3.1 Security Assessment Goals

The engagement was scoped to provide a security assessment of ZKC's source code. During the assessment, the security analysts aimed to answer questions such as:

- **zkVM Applications**
  - Is all host-provided input validated inside the guest?
  - Are environment configurations (e.g., library versions, external chain state) validated within the guest?
  - Is the zkVM configured correctly?
  - Do on-chain components enforce replay protection for submitted proofs?
- **Smart Contracts**
  - Are there centralization risks, such as excessive admin control or upgrade authority?
  - Can denial-of-service arise from gas exhaustion, logic flaws, or revert patterns?
  - Are mathematical operations (e.g., divide-before-multiply) precise and safe from rounding errors?
  - Does every state-modifying action emit an appropriate event?
  - Is access control correctly implemented for privileged functions?
  - Are user funds protected against accidental locking?
  - Are reentrancy protections consistently applied?
  - Are packing and hashing operations correct and collision-resistant?
  - Can voting or checkpoints be manipulated by large stakeholders or flash-loans?
  - Are replay attacks on transactions or proofs prevented?
  - Is the protocol resistant to front-running attacks in execution flows?
- **Custom Protocol Logic**
  - Are veZKC tokens correctly bound to accounts (soulbound vs. transferable)?
  - Do stake top-ups properly interact with delegation, vote history, and withdrawal periods?
  - Are rounding and math errors (esp. division vs. floor) prevented?
  - Is access control correctly enforced across all staking and delegation functions?
  - Is every event verified against the correct chain and contract?
  - Are only valid events of the correct type processed, avoiding contamination by irrelevant events?
  - Can malicious provers cause denial-of-service by making proof verification excessively costly?
  - Is the division of responsibility between ZK logic and on-chain checks clearly defined and enforced?
  - Can second pre-image or path-manipulation attacks compromise Merkle proofs?
  - Are Merkle proof paths fully validated to prevent unchecked traversal into the tree structure?

- Is the sum of reward/voting power consistent with staked funds minus pending withdrawals?
- Are emission schedules correctly enforced regardless of ZK logic, ensuring total supply never exceeds the schedule?
- Do staking, unstaking, delegation, and reward-claim workflows correctly enforce all documented invariants?

3.2 Security Assessment Methodology & Scope

Security Assessment Methodology. To address the questions above, the security assessment involved a combination of human experts and automated program analysis & testing tools. In particular, the security assessment was conducted with the aid of the following techniques:

- *Static analysis.* To identify potential common vulnerabilities, security analysts leveraged the open-source tool Slither. These tools are designed to find instances of common smart contract vulnerabilities, such as reentrancy and uninitialized variables.
- *Fuzzing/Property-based Testing*. Security analysts leveraged fuzz testing to determine if the protocol may deviate from the expected behavior. To do this, the desired behavior of the protocol was formulated as [V] specifications and then tested using Veridise's fuzzing framework OrCa to determine if a violation of the specification can be found. See Section 6 for more information.

Scope. The scope of this security assessment is limited to the following locations, which contain the smart contract implementation of the ZKC.

- `ZKC`:
  - `src/**.sol`
- `boundless`:
  - `contracts/src/povw/PovwMint.sol`
  - `crates/guest/povw/`
    - `mint-calculator/src/main.rs`
    - `src/mint_calculator.rs`

During the security assessment, the Veridise security analysts referred to the excluded files such as `PoVWAccounting.sol`, but assumed that they have been implemented correctly.

Methodology. Veridise security analysts reviewed the reports of previous audits for ZKC, inspected the provided tests, and read the ZKC documentation. They then began a review of the code assisted by both static analyzers and automated testing.

During the security assessment, the Veridise security analysts regularly met with the ZKC developers to ask questions about the code.

3.3 Classification of Vulnerabilities

When Veridise security analysts discover a possible security vulnerability, they must estimate its severity by weighing its potential impact against the likelihood that a problem will arise.

The severity of a vulnerability is evaluated according to the Table 3.1.

Table 3.1: Severity Breakdown.

	Somewhat Bad	Bad	Very Bad	Protocol Breaking
Not Likely	Info	Warning	Low	Medium
Likely	Warning	Low	Medium	High
Very Likely	Low	Medium	High	Critical

The likelihood of a vulnerability is evaluated according to the Table 3.2.

Table 3.2: Likelihood Breakdown

Not Likely	A small set of users must make a specific mistake
Likely	Requires a complex series of steps by almost any user(s) - OR - Requires a small set of users to perform an action
Very Likely	Can be easily performed by almost anyone

The impact of a vulnerability is evaluated according to the Table 3.3:

Table 3.3: Impact Breakdown

Somewhat Bad	Inconveniences a small number of users and can be fixed by the user
Bad	Affects a large number of people and can be fixed by the user - OR - Affects a very small number of people and requires aid to fix
Very Bad	Affects a large number of people and requires aid to fix - OR - Disrupts the intended behavior of the protocol for a small group of users through no fault of their own
Protocol Breaking	Disrupts the intended behavior of the protocol for a large group of users through no fault of their own

4 Trust Model

4.1 Operational Assumptions

In addition to assuming that any out-of-scope components behave correctly, Veridise analysts assumed the following properties held when modeling security for ZKC.

- Proof of verifiable work corresponds to actual proving work done. In particular, chaining the work log commit ID makes incremental proving techniques which reuse proving effort infeasible to apply.
- The out-of-scope functionality implemented by the `PoVWAccounting` and log-builder/updater circuits are implemented correctly, tracking verifiable work and globally preventing replays of the same proof task.

4.2 Privileged Roles

Roles. This section describes in detail the specific roles present in the system, and the actions each role is trusted to perform. During the review, Veridise analysts assumed that the role operators perform their responsibilities as intended. Protocol exploits relying on the below roles acting outside of their privileged scope are considered outside of scope.

- `ZKC.initialMinter1` and `ZKC.initialMinter2` may mint the initial one billion `ZKC` tokens.
- Users with the `ZKC.POVW_MINTER_ROLE` and `ZKC.STAKING_MINTER_ROLE` may mint rewards up to the amount of emissions allocated to minting and staking, respectively. These should *only* be granted to the `PoVWMint` and `StakingRewards` contracts, respectively.
- Users with the `ZKC.ADMIN_ROLE` or `veZKC.ADMIN_ROLE` may update the contracts.

Operational Recommendations. Highly-privileged, non-emergency operations should be operated by a multi-sig contract or decentralized governance system. These operations should be guarded by a timelock to ensure there is enough time for incident response. Highly-privileged, emergency operations should be tested in example scenarios to ensure the role operators are available and ready to respond when necessary.

Full validation of operational security practices is beyond the scope of this review. Users of the protocol should ensure they are confident that the operators of privileged keys are following best practices such as:

- Never storing a protocol key in plaintext, on a regularly used phone, laptop, or device, or relying on a custom solution for key management.
- Using separate keys for each separate function.
- Storing multi-sig keys in a diverse set of key management software/hardware services and geographic locations.

- Enabling 2FA for key management accounts. SMS should *not* be used for 2FA, nor should any account which uses SMS for 2FA. Authentication apps or hardware are preferred.
- Validating that no party has control over multiple multi-sig keys.
- Performing regularly scheduled key rotations for high-frequency operations.
- Securely storing physical, non-digital backups for critical keys.
- Actively monitoring for unexpected invocation of critical operations and/or deployed attack contracts.
- Regularly drilling responses to situations requiring emergency response such as pausing/unpausing.

5 Vulnerability Report

This section presents the vulnerabilities found during the security assessment. For each issue found, the type of the issue, its severity, location in the code base, and its current status (i.e., acknowledged, fixed, etc.) is specified. Table 5.1 summarizes the issues discovered:

Table 5.1: Summary of Discovered Vulnerabilities.

ID	Description	Severity	Status
V-ZKC-VUL-001	Reward cap can be bypassed using . . .	High	Fixed
V-ZKC-VUL-002	Duplicate events	Low	Fixed
V-ZKC-VUL-003	Unused and duplicate functionality	Warning	Fixed
V-ZKC-VUL-004	Non-atomic initialization	Warning	Fixed
V-ZKC-VUL-005	No address validation when initializing . . .	Warning	Fixed
V-ZKC-VUL-006	No address validation when initializing . . .	Warning	Fixed
V-ZKC-VUL-007	Dust left between rewards	Warning	Fixed
V-ZKC-VUL-008	Frontrunning can slow batch posting	Warning	Acknowledged
V-ZKC-VUL-009	Missing/incorrect documentation	Info	Fixed

5.1 Detailed Description of Issues

5.1.1 V-ZKC-VUL-001: Reward cap can be bypassed using multiple work log IDs

Severity	High	**Commit**	8bd70a8
Type	Logic Error	**Status**	Fixed
Location(s)	`crates/guest/povw/mint-calculator/src/main.rs:162-177`		
Confirmed Fix At	https://github.com/boundless-xyz/boundless/pull/1038		

When claiming PoVW rewards, provers are limited by a per-epoch reward cap. This reward cap is calculated as a fraction of their staked ZKC at a given epoch (currently one third of their stake). The mint-calculator guest is responsible for limiting the amount of tokens minted to a prover for a given epoch to this cap.

However, the mint-calculator guest currently calculates this cap *per recipient address*. Additionally, only updates to work log IDs in the work log filter will contribute to the calculated reward totals.

```
fn main() {
    // [VERIDISE] elided

    // Construct the mapping of calculated rewards, with the key as (epoch, recipient) pairs and
    // the value as a FixedPoint fraction indicating the portion of the PoVW epoch reward to assign
    let mut rewards_weights = BTreeMap::<U256, BTreeMap<Address, FixedPoint>>::new();
    let mut updates = BTreeMap::<Address, (B256, B256)>::new();
    for env in envs.0.values() { // [VERIDISE] Iterates through blocks
        // [VERIDISE] elided
            if !input.work_log_filter.includes(update_event.workLogId.into()) {
                continue; // [VERIDISE]: Filtered out events won't contribute to reward_weights
            }
        // [VERIDISE] elided
    }

    // [VERIDISE] elided

    // Calculate the rewards for each recipient by assigning the portion of each epoch rewards they
    // earned, capped by their max allowed reward in that epoch.
    let mut rewards = BTreeMap::<Address, U256>::new();
    // [VERIDISE] elided
    for (epoch, epoch_reward_weights) in rewards_weights {
        // [VERIDISE] elided
        for (recipient, weight) in epoch_reward_weights {
            // Calculate the maximum rewards, based on the povw value alone.
            let uncapped_reward = weight.mul_unwrap(epoch_emissions);

            // Get the reward cap for this recipient in the given epoch. Note that the reward cap
            // is determined at the end of the epoch.
```

```
            let reward_cap = zkc_rewards_contract
                .call_builder(&IZKCRewards::getPastPoVWRewardCapCall {
                    account: recipient, // [VERIDISE] the cap is calculated based on
    the recipient
                    timepoint: epoch_end_time,
                })
                .call();

            // Apply the cap and add the reward to the final mapping.
            let reward = U256::min(uncapped_reward, reward_cap);
            if reward > U256::ZERO {
                *rewards.entry(recipient).or_default() += reward;
            }
        }
    }
    // [VERIDISE] elided
}
```

Snippet 5.1: Abridged snippet of the mint-calculator guest application.

Combining these two facts, a prover can bypass their reward cap by using either multiple work log IDs, or multiple recipient addresses. This requires them to run the mint-calculator guest multiple times, reaching the reward cap in each run. Examples for both situations are presented in the following section.

Examples

- **Multiple work logs:**
  - A prover Penelope, with a cap of `N`, can exceed this limit by maintaining multiple work log IDs under the same recipient address. For example, Penelope produces `2N` work and records it across two logs `(w1, w2)`. She then runs the mint-calculator guest once filtered to `w1` and once to `w2`, minting `N` tokens each time. In total, she obtains `2N` tokens, bypassing her intended cap.
- **Multiple recipients:**
  - T
    wo provers can collude to evade their caps by alternating recipient addresses. For example, Penelope and Philip each have a cap of `N`. Penelope splits her work log updates so half credit her own address and half credit Philip's. When she runs the mint-calculator guest filtered to her work log, the output mints `2N` tokens (since both recipients independently qualify up to N). Philip repeats this with his own log, minting another `2N` tokens. Together they obtain `4N` tokens, exceeding the per-prover limits.

Impact Provers can bypass their intended reward cap by maintaining multiple work log IDs or by colluding with other provers.

Recommendation Enforce the reward cap on the work log ID. This would require a prover to stake additional tokens to maintain multiple work log IDs with positive reward caps, and prevent them from using multiple recipient addresses.

Developer Response The developers have updated the code so that the PoVW reward cap is tied to the work log ID instead of the recipient address in both the host and guest code. Each processed work log update then decreases the cap for that work log ID (regardless of the recipient address).

The implemented fix prevents both of the attack vectors suggested in this issue.

5.1.2 V-ZKC-VUL-002: Duplicate events

Severity	Low	**Commit**	f6f8f89
Type	Incorrect/Missing Events	**Status**	Fixed
Location(s)	`src/components/Staking.sol:101-126, 217-249`		
Confirmed Fix At	`https://github.com/boundless-xyz/zkc/pull/14`		

As shown in the below snippet, `initiateUnstake()` emits the `UnstakeInitiated`. However, it also calls the internal function `_initiateUnstakeAndCheckpoint()`, which emits a duplicate `UnstakeInitiated` event.

```
/// @inheritdoc IStaking
function initiateUnstake() external nonReentrant {
    // [VERIDISE] elided....

    // Mark as withdrawing and checkpoint (powers drop to 0)
    _initiateUnstakeAndCheckpoint(tokenId);

    uint256 withdrawableAt = block.timestamp + Constants.WITHDRAWAL_PERIOD;
    emit UnstakeInitiated(tokenId, msg.sender, withdrawableAt);
}
```

Snippet 5.2: Portion of the `initiateUnstake` function which calls `_initiateUnstakeAndCheckpoint` and emits an `UnstakeInitiated` event

```
function _initiateUnstakeAndCheckpoint(uint256 tokenId) internal {
    // [VERIDISE] elided....

    uint256 withdrawableAt = newStake.withdrawalRequestedAt + Constants.
    WITHDRAWAL_PERIOD;
    emit UnstakeInitiated(tokenId, owner, withdrawableAt);
}
```

Snippet 5.3: Portion of the `_initiateUnstakeAndCheckpoint` method which emits an `UnstakeInitiated` event

A similar issue exists for the `StakeAdded` event, both `_addStakeAndCheckpoint()` and `_addToStake()` emit the same event.

Impact Off-chain listeners may mistakenly credit users with double the amount they add to stake. Similarly, off-chain listeners may penalize unstaking users by subtracting the value from their stake twice.

Recommendation Only emit each event once during execution.

Developer Response The additional `StakeAdded` was removed from `_addToStake()`, and the extra `UnstakeInitiated` was removed from `initiateUnstake()`.

5.1.3 V-ZKC-VUL-003: Unused and duplicate functionality

Severity	Warning	**Commit**	f6f8f89
Type	Maintainability	**Status**	Fixed
Location(s)	`src/` ▶ `ZKC.sol:94` ▶ `components/Storage.sol:35` ▶ `interfaces/` • `IStaking.sol:10-20` • `IVotes.sol:11` ▶ `libraries/` • `[...]/Checkpoints.sol:122, 134, 142, 149, 166, 221-229, 247-250, 277-283, 295-298, 322-325, 370-373, 424-432` • `RewardPower.sol:25` • `StakeManager.sol:17-23` • `Supply.sol:32, 55, 57-64, 77` • `VotingPower.sol:55` ▶ `rewards/StakingRewards.sol:43` ▶ `veZKC.sol:43`			
Confirmed Fix At	`https://github.com/boundless-xyz/zkc/pull/16`, `https://github.com/boundless-xyz/zkc/pull/16`		

Impact The following errors and internal functions are unused:

1. `src/components/`
   - ▶ `Storage.sol`: `_ownedTokens`, `_ownedTokensIndex`
2. `src/interfaces/`
   - ▶ `IStaking.sol`: `UserAlreadyHasActivePosition`, `CannotAddToWithdrawingPosition`, `WithdrawalAlreadyInitiated`, `WithdrawalNotInitiated`, and `WithdrawalPeriodNotComplete`
   - ▶ `IVotes.sol`: `IVotes.NotImplemented`
3. `src/libraries/`
   - ▶ `Checkpoints.sol`: `Checkpoints.checkpoint()`, `Checkpoints.getUserPoint()`, `Checkpoints.getGlobalPoint()`, `Checkpoints.getUserEpoch()`, `Checkpoints.getGlobalEpoch()`. Importantly, in `Checkpoints.checkpoint()`, if `newStake.amount == 0` then `userNewPoint.updatedAt` is set to the zero-timestamp. If `checkpoint()` is not removed, this surprising behavior may trigger bugs in the future.
   - ▶ `VotingPower.sol`: `VotingPower.getTotalSupply()`

The following code locations duplicate functionality implemented elsewhere:

1. `src/veZKC.sol`:
   - `veZKC.initialize()`: This function grants the `DEFAULT_ADMIN_ROLE` rather than the `veZKC.ADMIN_ROLE`, which is set to the same value.
2. `src/ZKC.sol`:
   - `ZKC.initialize()`: This function grants the `DEFAULT_ADMIN_ROLE` rather than the `ZKC.ADMIN_ROLE`, which is set to the same value.
3. `src/libraries/`
   - `RewardPower.sol`:
     - `getPastStakingRewards()`, `getTotalStakingRewards()`, and `getPastTotalStakingRewards()` all repeat the same computation on `Checkpoints.Point` to compute the reward power.
   - `StakeManager.sol`:
     - Each custom `error` is a redefinition of an `error` with the same name in `IStaking.sol`.
   - `Supply.sol`:
     - The implementation assumes `Y8_R_PER_EPOCH == FINAL_R_PER_EPOCH`. The definition of `FINAL_R_PER_EPOCH` is copied from `Y8_PER_EPOCH` rather than explicitly setting `FINAL_R_PER_EPOCH = Y8_PER_EPOCH`.
     - The definition of `SUPPLY_YEAR_0` is copied from the definition of `INITIAL_SUPPLY`, rather than setting `SUPPLY_YEAR_0 = INITIAL_SUPPLY` explicitly.
     - `Supply.getGrowthFactor()`: lines 57-64 of this function inline `_getGrowthFactorForYear()`.
     - Lines 55 and 77 inline the `getYearForEpoch()` function.
4. `src/rewards/StakingRewards.sol`:
   - `StakingRewards.initialize()`: This function grants the `DEFAULT_ADMIN_ROLE` rather than the `StakingManager.ADMIN_ROLE`, which is set to the same value.
5. `src/libraries/Checkpoints.sol`:
   - The blocks of code in L221-229, L424-432 and L277-283 all duplicate the same logic for updating the global point history. The block in L277-283 also flips the order of the blocks, making it inconsistent with the rest.
   - The logic for retrieving a user's latest checkpoint or creating a new one is duplicated in L247-250, L295-298, L322-325 and L370-373.

Recommendation Remove the unused functionality, and remove the reimplemented functionality.

Developer Response The developers have applied all of the recommended fixes.

5.1.4 V-ZKC-VUL-004: Non-atomic initialization

Severity	Warning	**Commit**	f6f8f89
Type	Access Control	**Status**	Fixed
Location(s)	`src/ZKC.sol:98-100`		
Confirmed Fix At	`https://github.com/boundless-xyz/zkc/pull/26`		

The supply of ZKC increases at regular 2-day intervals, called "epochs". At the beginning of epoch 0, an initial supply of one billion ZKC are present. At the start of each epoch thereafter, additional ZKC are made available according to a fixed emission schedule.

Epoch 0 does not start at contract creation. Instead, it is started by calling `initializeV2()`, shown in the below code snippet. However, there is no access control on this function. The `reinitializer(2)` modifier ensures it is called at most once, and not before `initialize()`. It does not prevent malicious users from calling `initializeV2()` before the deployers are ready for the epoch to begin.

```
function initializeV2() public reinitializer(2) {
    epoch0StartTime = block.timestamp;
}
```

Snippet 5.4: Definition of `ZKC.initializeV2()`

Impact Epoch 0, and consequently the entire ZKC emission schedule, may start before intended.

For example, if initial mints are performed before `epoch0StartTime` and a prover comes online before other provers, they may use this function to begin pow-based token distribution while they still have a competitive advantage.

Recommendation As recommended in Epoch times incorrect before 'initializeV2()', initialize the `epoch0StartTime` to a time far in the future. Additionally, add access control so only trusted users can call `initializeV2()`.

Developer Response The developers have acknowledged the issue, and stated:

> We have already deployed V1, and are planning to upgrade to V2 atomically. We intend to call initializeV2 atomically during upgrade, using OpenZeppelin's provided library function. See upgrade script here.

To resolve the problem, they now initialize the `epoch0StartTime` to be `type(uint256).max` upon `initializeV2()` and add an access-controlled `initializeV3()` function which starts the 0th epoch and can only be called once. All functions which require or compute an epoch start/end time now revert until the `epoch0StartTime` has been set to a non-default value.

Updated Veridise Response It may still be technically possible to stake before epoch 0, however this should only affect functions like `getPastVotes()`, `getPastTotalSupply()`, and their rewards variants.

Updated Developer Response This is a good point, but we agree that it should be safe. Rewards still cannot be minted. Since the reward power is flat and does not grow you do not gain any advantage to staking before epochs have started.

We do not expect the mentioned methods to be used, and do not see any risk to having them return valid values before epoch 0 either.

5.1.5 V-ZKC-VUL-005: No address validation when initializing contracts

Severity	Warning	**Commit**	f6f8f89
Type	Data Validation	**Status**	Fixed
Location(s)	`src/` ▶ `ZKC.sol:90-94` ▶ `rewards/StakingRewards.sol:42-45` ▶ `veZKC.sol:48-49`		
Confirmed Fix At	`https://github.com/boundless-xyz/zkc/pull/17`		

The `initialize()` function in several contracts (`ZKC`, `veZKC`, and `StakingRewards`) sets several configuration parameters such as token and admin addresses, but fails to validate them. Specifically:

- The `ZKC` contract does not validate:
  - The `_initialMinter1` and `_initialMinter2` addresses, which will be given the ability to mint tokens from the initial supply.
  - The `_owner` address, which is given the admin role.
- The `veZKC` contract does not validate:
  - The address of the ZKC token, `zkcTokenAddress`, which it will use when performing token transfers for staking/unstaking operations.
  - The `_admin` address, which is given the admin role.
- The `StakingRewards` contract does not validate:
  - The address of the ZKC and veZKC tokens, `_zkc` and `_veZKC`. These are used to calculate rewards for users, and to request ZKC token mints.
  - The `_admin` address, which is given the admin role.

Since this is an `initializer` and can only be called once, incorrect input (e.g., a zero address) will brick the contract or render its functionality unusable until the contract is upgraded.

Impact If any of the token addresses or minter addresses are null, critical parts of the protocol will not work (e.g. minting the initial supply of tokens). Additionally, if the admin addresses are set to be zero, the contracts will no longer be upgradeable.

Recommendation Add sanity checks to the initializer functions to check that all addresses are non-null.

Developer Response The developers implemented the recommendation.

5.1.6 V-ZKC-VUL-006: No address validation when initializing PovwMint

Severity	Warning	**Commit**	8bd70a8
Type	Data Validation	**Status**	Fixed
Location(s)	`contracts/src/povw/PovwMint.sol:89-93`		
Confirmed Fix At	d1cd74a		

The constructor of the `PovwMint` contract takes several addresses as input. Notably, it expects the address of an `IRiscZeroVerifier`, which will be used to verify the seal of the mint-calculator guest; it also expects the addresses of the ZKC and veZKC tokens, which will be used to verify the journal from the mint-calculator guest. However, there are no sanity checks performed on these addresses (e.g. to check that they are non-zero).

```
/// @notice Mint tokens as a reward for verifiable work.
function mint(bytes calldata journalBytes, bytes calldata seal) external {
        // Verify the mint is authorized by the mint calculator guest.
        VERIFIER.verify(seal, MINT_CALCULATOR_ID, sha256(journalBytes));
        MintCalculatorJournal memory journal = abi.decode(journalBytes, (
    MintCalculatorJournal));
        if (!Steel.validateCommitment(journal.steelCommit)) {
            revert InvalidSteelCommitment();
        }
        if (journal.povwAccountingAddress != address(ACCOUNTING)) {
            revert IncorrectSteelContractAddress({
                expected: address(ACCOUNTING),
                received: journal.povwAccountingAddress
            });
        }
        if (journal.zkcAddress != address(TOKEN)) {
            revert IncorrectSteelContractAddress({expected: address(TOKEN), received:
     journal.zkcAddress});
        }
        if (journal.zkcRewardsAddress != address(TOKEN_REWARDS)) {
            revert IncorrectSteelContractAddress({expected: address(TOKEN_REWARDS),
    received: journal.zkcRewardsAddress});
        }
    // [VERIDISE] elided
}
```

Snippet 5.5: Snippet from the `mint()` function, which verifies the addresses used by the mint-calculator guest.

Impact If the address of the verifier or any of the tokens is set to be null the contract will not be able to verify valid proofs.

Recommendation Add sanity checks to the constructor of the `PovwMint` contract.

Developer Response The developers implemented the recommendation, and checked both `PovwAccounting` and `PovwMint` for zero-addresses.

5.1.7 V-ZKC-VUL-007: Dust left between rewards

Severity	Warning	**Commit**	f6f8f89
Type	Rounding Error	**Status**	Fixed
Location(s)	`src/ZKC.sol:161-170`		
Confirmed Fix At	`https://github.com/boundless-xyz/zkc/pull/18`		

The `getTotal*EmissionsAtEpochStart()` functions are used to determine the maximum amount which may be distributed due to either PoVW or staking rewards from the protocol start up to the start of the current epoch. This is computed as a fraction of the total ZKC emissions up to this point, with 75% of the rewards going to PoVW rewards and 25% to staking. These are computed in fixed precision as shown in the below snippet.

```
function getTotalPoVWEmissionsAtEpochStart(uint256 epoch) public pure returns (
    uint256) {
    uint256 totalEmissions = getSupplyAtEpochStart(epoch) - INITIAL_SUPPLY;
    return (totalEmissions * POVW_ALLOCATION_BPS) / BASIS_POINTS;
}

function getTotalStakingEmissionsAtEpochStart(uint256 epoch) public pure returns (
    uint256) {
    uint256 totalEmissions = getSupplyAtEpochStart(epoch) - INITIAL_SUPPLY;
    return (totalEmissions * STAKING_ALLOCATION_BPS) / BASIS_POINTS;
}
```

Snippet 5.6: Caps on emissions from PoVW and staking rewards, respectively.

Both computations round down. This means there may be some small amount of ZKC emissions which are inaccessible to both reward pools.

Impact There may be dust amounts of ZKC which are inaccessible to any user.

Recommendation Round up one of the computations, and round the other down. The same comment applies to `getPoVWEmissionsForEpoch()` and `getStakingEmissionsForEpoch()`.

Developer Response The developers now round up the computation when computing PoVW emissions.

5.1.8 V-ZKC-VUL-008: Frontrunning can slow batch posting

Severity	Warning	**Commit**	8bd70a8
Type	Frontrunning	**Status**	Acknowledged
Location(s)	`crates/guest/povw/mint-calculator/src/main.rs`		
Confirmed Fix At	N/A		

Anyone may post updates from a work log to mint rewards accordingly. When someone does this, any other proofs being generated will be invalidated. This is because `PoVWMint.mint()` checks that the initial work log commit matches the latest one used to mint rewards on-chain for the specified work log.

If a user has waited for hundreds of epochs to claim rewards, this may allow an attacker to dramatically slow the rate at which the user may claim rewards. For example, if a user has 100 pending epochs of rewards, they may make a batch claim proof. The attacker may prepare a proof which claims rewards for only one of those epochs. If the attacker can post their proof before the user, then the batch claim will fail.

Impact Attackers can force users to claim rewards one epoch at a time. This may slow reward claiming during critical epochs related to votes.

Recommendation Allow workers to (optionally) whitelist which users may claim rewards on their work logs.

Developer Response The developers have acknowledged the issue and stated:

> We have considered this scenario and decided that it is an acceptable level of risk.
>
> As noted, the attacker would be able to slow the reward claim process to 1 epoch per transaction, preventing the user from claiming rewards for e.g. the last 30 days in one transaction. 1 epoch is currently set to 2 days. As a result, this would mean that the reward claim could take up to e.g. 15 transaction for 30 days. Each transaction has an associated proof cost. This slowdown is not trivial, but also does not prevent the prover from eventually claiming their full rewards.
>
> As mitigating factors, the attacker would have to produce a proof and submit a transaction to front-run each update. This reduces the usefulness of this attack in two ways:
>
> 1. The proofs must be prepared ahead of time, rather than upon seeing the reward claim transaction in the mempool, assuming the attacker cannot delay the mempool transaction for significant periods of time.
> 2. The attack has transaction costs onchain for each round.

5.1.9 V-ZKC-VUL-009: Missing/incorrect documentation

Severity	Info	**Commit**	f6f8f89
Type	Maintainability	**Status**	Fixed
Location(s)	`src/` ▶ `libraries/` • `Checkpoints.sol:14-16` • `StakeManager.sol:42-43` • `Supply.sol:189-194` ▶ `rewards/StakingRewards.sol:57-59`		
Confirmed Fix At	`https://github.com/boundless-xyz/zkc/pull/15`		

The following code locations are missing documentation, or could benefit from additional elaboration.

- `src/libraries/`
  - `Checkpoints.sol`:
    - The comments on the `votingAmount` and `rewardAmount` fields of the `Point` struct indicate that voting and reward power are counted from delegation and the user's stake. However, the user's stake only contributes to these if they aren't delegating them to another user.

```
/// @notice Amount counting toward voting power (own stake + delegated votes)
uint256 votingAmount;
/// @notice Amount counting toward reward power (own stake + delegated rewards)
uint256 rewardAmount;
```

  - `StakeManager.sol`:
    - The `StakeManager.addToStake()` function indicates that stake may be added while a withdrawal is pending. However, adding stake to a position which is pending withdrawal is disallowed.

```
withdrawalRequestedAt: 0 // Reset withdrawal when adding stake
```

  - `Supply.sol`:
    - The `CACHE_PREFIX` (shown below) is used to avoid potential transient storage collisions in future versions of the protocol. The 12-byte prefix appears to be randomly or pseudo-randomly selected. However, this process is not clearly documented. If the prefix comes from a hash of a specific string, this may lead to unexpected collisions in future versions. The source of this prefix should be documented to ensure this does not become an issue.

```
    /// @notice Supply values for epochs are cached, to enable efficient batch claims of epochs.
    /// @dev This is a transient storage cache, so it is not persisted across blocks.
    ///      NOTE: We do not need to clear the cache after use, as supply values are deterministic.
```

```
4     /// @dev Apply a prefix to reduce risk of collisions with future
      tstore features.
5     ///      Leaves 20 bytes for epoch (max epoch: 2^160 - 1).
6     bytes32 private constant CACHE_PREFIX = 0
      x5A4B43454D495353494F4E5300000000000000000000000000000000000000
```

- `src/rewards/StakingRewards.sol`:
  - The `calculateRewards()` function allows users to supply duplicate and current or future epochs in the provided array of epochs. Current/future epochs will correspond to a zero reward entry in the output array. Duplicate epochs will each list the same reward amount. Third-party integrators may misuse this method, allowing users to falsely inflate or deflate rewards accounted on third-party platforms. Consider documenting the differences between this and `claimRewards()`, which enforces uniqueness of epochs and only accepts past epochs. Alternatively, enforce the same checks.

Impact Missing documentation may make the code more difficult to understand in the future, leading to potential future errors.

Recommendation Update the documentation.

Developer Response The developers implemented the recommendation.

6 Fuzz Testing

6.1 Methodology

One of the goals of the security assessment was to fuzz test ZKC with a few key goals in mind, namely:

- Check that the different workflows, such as unstaking, delegating and topping up existing positions, could not be called in unexpected ways.
- After execution of such functions, the state of the contract is modified as expected.
- The protocol maintains proper accounting of accumulated token emissions, staked tokens, and voting/reward power amounts.

The Veridise security analysts used the OrCa tool to fuzz the project. The analysts captured the intended behavior of the system by writing invariants–logical formulas which should hold after each transaction. The invariants were encoded as statements in the [V] specification language and passed to the OrCa tool, together with the project program code.

Minor modifications were made to the project program code to enable proper deployment and add additional information that could be leveraged by OrCa. These changes had no impact on the existing logic of the contracts.

6.2 Properties Fuzzed

Table 6.1 describes the fuzz-tested invariants. The second column describes the invariant informally in English, and the third shows the total amount of compute time spent fuzzing this property. The last column indicates the number of bugs identified when fuzzing the invariant.

The first set of invariants were written targeting specific functions of ZKC. Specifically, the targets functions were `stake()`, `add_to_stake()`, `delegate()`, `delegateRewards()`, `initiateUnstake()`, and `completeUnstake()`. These invariants checked expected pre- and post-conditions that were expected when being used properly. Additionally, further invariants were written that checked global properties that were expected to hold at any point in the project's lifespan. These invariants targeted the accounting of tokens, staked balances and voting/reward power. The individual invariants are detailed in **??**.

The Veridise team devoted a total of 217 compute-hours to fuzzing this protocol, identifying a total of 0 bugs.

Table 6.1: Invariants Fuzzed.

Specification	Invariant	Minutes Fuzzed	Bugs Found
V-ZKC-SPEC-001	Claimed rewards never exceed emission . . .	730	1
V-ZKC-SPEC-002	Claimed total supply matches expected	310	0
V-ZKC-SPEC-003	Completing an unstake updates state	730	0
V-ZKC-SPEC-004	Current or future epochs are unusable	670	0
V-ZKC-SPEC-005	Initiating an unstake meets preconditions	730	0
V-ZKC-SPEC-006	Initiating an unstake updates state correctly	730	0
V-ZKC-SPEC-007	No underflows on reward delegation	60	0
V-ZKC-SPEC-008	No underflows on unstaking	60	0
V-ZKC-SPEC-009	No underflows on vote delegation	120	0
V-ZKC-SPEC-010	Nop reward delegation has no change	670	0
V-ZKC-SPEC-011	Nop vote delegation has no change	670	0
V-ZKC-SPEC-012	One position per owner	310	0
V-ZKC-SPEC-013	Owned tokens correspond to a positive stake	310	0
V-ZKC-SPEC-014	Past rewards are unchangeable	670	0
V-ZKC-SPEC-015	Reward delegation changes state correctly	670	0
V-ZKC-SPEC-016	Reward delegation meets preconditions	670	0
V-ZKC-SPEC-017	Reward power matches history	60	0
V-ZKC-SPEC-018	Staking rewards cannot be claimed more . . .	670	0
V-ZKC-SPEC-019	Staking updates state correctly	670	1
V-ZKC-SPEC-020	Upping stake by token ID updates state correctly	670	0
V-ZKC-SPEC-021	Upping stake updates state correctly	730	1
V-ZKC-SPEC-022	Vote delegation changes state correctly	670	0
V-ZKC-SPEC-023	Vote delegation meets preconditions	670	0
V-ZKC-SPEC-024	ZKC Emission rate is respected	790	1

6.3 Detailed Description of Fuzzed Specifications

6.3.1 V-ZKC-SPEC-001: Claimed rewards never exceed emission allocation

Minutes Fuzzed	730	Bugs Found	1

Natural Language Description Claimed rewards never exceed available emissions for the current epoch: both PoVW and staking claimed amounts are bounded by their respective total emissions at the epoch start.

Formal Specification

```
vars: ZKC zkc
inv: (
    # never have more rewards than the limits
    (zkc.poVWClaimed() <= zkc.getTotalPoVWEmissionsAtEpochStart(zkc.getCurrentEpoch()))
    && (zkc.stakingClaimed() <= zkc.getTotalStakingEmissionsAtEpochStart(zkc.getCurrentEpoch()))
)
```

6.3.2 V-ZKC-SPEC-002: Claimed total supply matches expected

Minutes Fuzzed	310	Bugs Found	0

Natural Language Description Claimed total supply equals the initial supply minus remaining initial minter balances plus all minted PoVW and staking rewards. This ties the accounting of claims directly to issuance and reward minting.

Formal Specification

```
vars: ZKC zkc
inv:
    # The total claimed supply must be
    zkc.claimedTotalSupply()
    =
    # The total allocated initial supply
    zkc.INITIAL_SUPPLY()
    # Minus the parts that haven't been minted yet
    - zkc.initialMinter1Remaining()
    - zkc.initialMinter2Remaining()
    # Plus the rewards minted
    + fsum{zkc.mintPoVWRewardsForRecipient(r, amount)}(amount)
    + fsum{zkc.mintStakingRewardsForRecipient(r, amount)}(amount)
```

6.3.3 V-ZKC-SPEC-003: Completing an unstake updates state

Minutes Fuzzed	730	Bugs Found	0

Natural Language Description Completing an unstake (modeled over the call shown) removes the active position, requires an initiated unstake with withdrawal period elapsed, transfers the staked amount back to the user's ZKC balance, and zeroes the staked amount.

Formal Specification

```
vars: veZKC vez, ZKC zkc
inv: (
    # Must have an active position
    old(vez.getActiveTokenId(sender)) != 0
    # User no longer has an active position
    && vez.getActiveTokenId(sender) = 0
    # Must have initiated unstake
    && vez.getStakedAmountAndWithdrawalTime(sender)[1] > 0
    # Withdrawal period must have elapsed
    && timestamp >= vez.getStakedAmountAndWithdrawalTime(sender)[1] + vez.
    WITHDRAWAL_PERIOD
    # ZKC balance increases after withdrawal
    && zkc.balanceOf(sender) = old(zkc.balanceOf(sender)) + old(vez.
    getStakedAmountAndWithdrawalTime(sender))[0]
    # User no longer has any stake
    && vez.getStakedAmountAndWithdrawalTime(sender)[0] = 0
) over vez.initiateUnstake()
```

6.3.4 V-ZKC-SPEC-004: Current or future epochs are unusable

Minutes Fuzzed	670	Bugs Found	0

Natural Language Description Disallows claims for the current or any future epoch. After executing any rewards function, it must not be the case that a user has claimed rewards for `epoch >= currentEpoch`.

Formal Specification

```
vars: StakingRewards rewards, uint256 epoch, address user
# It is never the case that
spec: []!(
    # After executing some rewards function
    finished(rewards.*,
        # user has claimed rewards for a current or future epoch
        (epoch >= rewards.getCurrentEpoch())
        &&
        (rewards.hasUserClaimedRewards(user, epoch))
    )
)
```

6.3.5 V-ZKC-SPEC-005: Initiating an unstake meets preconditions

Minutes Fuzzed	730	Bugs Found	0

Natural Language Description Initiating an unstake requires an active position, preserves the token ID and staked amount, and must be the first unstake request (withdrawal time was 0).

Formal Specification

```
vars: veZKC vez
inv: (
    # Must have an active position
    old(vez.getActiveTokenId(sender)) != 0
    # Token ID doesn't change
    && old(vez.getActiveTokenId(sender)) = vez.getActiveTokenId(sender)
    # Stake doesn't change
    && old(vez.getStakedAmountAndWithdrawalTime(sender))[0] = vez.
    getStakedAmountAndWithdrawalTime(sender)[0]
    # Must be first unstaking request
    && old(vez.getStakedAmountAndWithdrawalTime(sender)[1]) = 0
) over vez.initiateUnstake()
```

6.3.6 V-ZKC-SPEC-006: Initiating an unstake updates state correctly

Minutes Fuzzed	730	Bugs Found	0

Natural Language Description Initiating an unstake reduces the user's voting and reward power by their full staked amount, requires no active delegation (self-delegated), and leaves ZKC balance unchanged.

Formal Specification

```
vars: veZKC vez, ZKC zkc
inv: (
    # # Voting/reward power decreases
    vez.getVotes(sender) = old(vez.getVotes(sender)) - vez.
    getStakedAmountAndWithdrawalTime(sender)[0]
    && vez.getStakingRewards(sender) = old(vez.getStakingRewards(sender)) - vez.
    getStakedAmountAndWithdrawalTime(sender)[0]
    # Can't be delegating
    && old(vez.delegates(sender)) = sender
    && old(vez.rewardDelegates(sender)) = sender
    # Balance doesn't change
    && zkc.balanceOf(sender) = old(zkc.balanceOf(sender))
) over vez.initiateUnstake()
```

6.3.7 V-ZKC-SPEC-007: No underflows on reward delegation

Minutes Fuzzed	60	Bugs Found	0

Natural Language Description When changing reward delegation, the previous reward delegate must have reward power at least equal to the user's staked amount, avoiding underflow on decrement.

Formal Specification

```
vars: veZKC vez
inv: (
    # The old delegate must have at least as much reward power as the user's stake
    old(vez.getStakingRewards(vez.rewardDelegates(sender))) >= vez.
    getStakedAmountAndWithdrawalTime(sender)[0]
# Whenever the user changes their reward delegate
) over vez.delegateRewards
```

6.3.8 V-ZKC-SPEC-008: No underflows on unstaking

Minutes Fuzzed	60	Bugs Found	0

Natural Language Description Before initiating an unstake, the user's voting and reward power must be at least their staked amount, preventing underflows when power is reduced.

Formal Specification

```
vars: veZKC vez
inv: (
    # The user must have had at least as much voting power as their staked amount
    old(vez.getVotes(sender)) >= old(vez.getStakedAmountAndWithdrawalTime(sender))[0]
    &&
    # And at least as much reward power as their staked amount
    old(vez.getStakingRewards(sender)) >= old(vez.getStakedAmountAndWithdrawalTime(
    sender)[0])
# Whenever the user unstakes
) over vez.initiateUnstake()
```

6.3.9 V-ZKC-SPEC-009: No underflows on vote delegation

Minutes Fuzzed	120	Bugs Found	0

Natural Language Description When changing voting delegation, the previous voting delegate must have voting power at least equal to the user's staked amount, avoiding underflow on decrement.

Formal Specification

```
vars: veZKC vez
inv: (
    # The old delegate must have at least as much voting power as the user's stake
    old(vez.getVotes(vez.delegates(sender))) >= vez.getStakedAmountAndWithdrawalTime(
    sender)[0]
# Whenever the user changes their voting delegate
) over vez.delegate
```

6.3.10 V-ZKC-SPEC-010: Nop reward delegation has no change

Minutes Fuzzed	670	Bugs Found	0

Natural Language Description If the rewards delegate is unchanged, the delegate's reward power does not change.

Formal Specification

```
vars: veZKC vez, ZKC zkc, uint256 stakeAmount, address oldDelegate
inv: (
    # If the new delegate is the same, no change
    ((old(vez.rewardDelegates(sender)) = newDelegate) ==>
            vez.getStakingRewards(newDelegate) = old(vez.getStakingRewards(
    newDelegate)))
    && ((old(vez.rewardDelegates(sender)) = newDelegate) ==>
            vez.getStakingRewards(old(vez.rewardDelegates(sender))) = old(vez.
    getStakingRewards(old(vez.rewardDelegates(sender)))))
) over vez.delegateRewards(newDelegate)
```

6.3.11 V-ZKC-SPEC-011: Nop vote delegation has no change

Minutes Fuzzed	670	Bugs Found	0

Natural Language Description If the voting delegate is unchanged, the delegate's voting power does not change.

Formal Specification

```
vars: veZKC vez, ZKC zkc, address oldDelegate
inv: (
    # If the new delegate is the same, no change
    ((old(vez.delegates(sender)) = newDelegate) ==>
            vez.getVotes(newDelegate) = old(vez.getVotes(newDelegate)))
    && ((old(vez.delegates(sender)) = newDelegate) ==>
            vez.getVotes(old(vez.delegates(sender))) = old(vez.getVotes(old(vez.
    delegates(sender)))))
) over vez.delegate(newDelegate)
```

6.3.12 V-ZKC-SPEC-012: One position per owner

Minutes Fuzzed	310	Bugs Found	0

Natural Language Description Enforces uniqueness of ownership: two distinct veZKC token IDs cannot be owned by the same address.

Formal Specification

```
vars: veZKC vez, uint256 t1, uint256 t2
inv:!(
    # 2 different token IDs cannot have the same owner
    t1 != t2 && vez.ownerOf(t1) = vez.ownerOf(t2)
)
```

6.3.13 V-ZKC-SPEC-013: Owned tokens correspond to a positive stake

Minutes Fuzzed	310	Bugs Found	0

Natural Language Description Any owned token must correspond to an owner with a positive staked amount; unowned tokens are allowed.

Formal Specification

```
vars: veZKC vez
inv: (
    # Either the token is unowned, or the owner has some tokens staked
    ret = 0 || vez.getStakedAmountAndWithdrawalTime(ret)[0] > 0
) over vez.ownerOf
```

6.3.14 V-ZKC-SPEC-014: Past rewards are unchangeable

Minutes Fuzzed	670	Bugs Found	0

Natural Language Description Past rewards are immutable: for epochs before the previous current epoch, `calculateRewards(user, [epoch])[0]` is unchanged. Current and future epochs always yield 0. Also enforces that ZKC and StakingRewards epochs are aligned.

Formal Specification

```
vars: StakingRewards rewards, uint256 epoch, address user, ZKC zkc
inv: (
    # past reward values should never change
    (rewards.getCurrentEpoch() = 0)
    ||
    (
        epoch < old(rewards.getCurrentEpoch())
        && old(rewards.calculateRewards(user, [epoch]))[0] = rewards.calculateRewards
    (user, [epoch])[0]
    )
    ||
    # current/future rewards should always be zero
    (
        epoch >= old(rewards.getCurrentEpoch())
        && old(rewards.calculateRewards(user, [epoch])[0] = 0)
    )
# ZKC epoch matches reward epoch
) && zkc.getCurrentEpoch() = rewards.getCurrentEpoch()
```

6.3.15 V-ZKC-SPEC-015: Reward delegation changes state correctly

Minutes Fuzzed	670	Bugs Found	0

Natural Language Description When changing the rewards delegate to a different address, the old delegate's reward power decreases by the user's staked amount, and the new delegate's reward power increases by that amount.

Formal Specification

```
vars: veZKC vez, ZKC zkc, address oldDelegate
inv: (
    # Old delegate power decreases if the new one is different
    ((old(vez.delegates(sender)) != newDelegate) ==>
            vez.getVotes(old(vez.delegates(sender))) = old(vez.getVotes(old(vez.
    delegates(sender)))) - old(vez.getStakedAmountAndWithdrawalTime(sender)[0]))
    # New delegate power increseases if different from old one
    && ((old(vez.delegates(sender)) != newDelegate) ==>
            vez.getVotes(newDelegate) = old(vez.getVotes(newDelegate)) + old(vez.
    getStakedAmountAndWithdrawalTime(sender)[0]))
) over vez.delegate(newDelegate)
```

6.3.16 V-ZKC-SPEC-016: Reward delegation meets preconditions

Minutes Fuzzed	670	Bugs Found	0

Natural Language Description Changing the rewards delegate has the same preservation properties: requires an active position, keeps token ID and staked amount unchanged, and requires no ongoing withdrawal.

Formal Specification

```
vars: veZKC vez, ZKC zkc, uint256 stakeAmount, address oldDelegate
inv: (
    # Must have an active position
    old(vez.getActiveTokenId(sender)) != 0
    # Token ID doesn't change
    && old(vez.getActiveTokenId(sender)) = vez.getActiveTokenId(sender)
    # Must have amount staked (implied by having an active position)
    && old(vez.getStakedAmountAndWithdrawalTime(sender)[0]) > 0
    # Staked amount doesn't change
    && vez.getStakedAmountAndWithdrawalTime(sender)[0] = old(vez.
    getStakedAmountAndWithdrawalTime(sender)[0])
    # Cannot be withdrawing
    && old(vez.getStakedAmountAndWithdrawalTime(sender)[1]) = 0
    && vez.getStakedAmountAndWithdrawalTime(sender)[1] = 0
) over vez.delegateRewards(newDelegate)
```

6.3.17 V-ZKC-SPEC-017: Reward power matches history

Minutes Fuzzed	60	Bugs Found	0

Natural Language Description Total reward power equals the sum of all stake and stake top-up amounts minus the amounts from positions that initiated unstake. Total voting power equals total reward power.

Formal Specification

```
vars: veZKC vez, ZKC zkc
inv:
    # The total reward power must be equal to
    vez.getTotalStakingRewards()
    =
    # The sum of all stakes
    fsum{vez.stake(amount)}(amount) +
    fsum{vez.stakeWithPermit(amount, d, v, r, s)}(amount) +
    # Plus the sum of all stake top-ups
    fsum{vez.addToStake(amount)}(amount) +
    fsum{vez.addToStakeWithPermit(amount, d, v, r, s)}(amount) +
    fsum{vez.addToStakeByTokenId(t, amount)}(amount) +
    fsum{vez.addToStakeWithPermitByTokenId(t, amount, d, v, r, s)}(amount) -
    # Minus all the positions that have requested to be unstaked
    fsum{vez.initiateUnstake()}(vez.getStakedAmountAndWithdrawalTime(sender)[0])
    &&
    # And the total voting power must be the same as the total reward power
    vez.getTotalVotes()
    =
    vez.getTotalStakingRewards()
```

6.3.18 V-ZKC-SPEC-018: Staking rewards cannot be claimed more than once

Minutes Fuzzed	670	Bugs Found	0

Natural Language Description Past rewards are immutable: for epochs before the previous current epoch, `calculateRewards(user, [epoch])[0]` is unchanged. Current and future epochs always yield 0. Also enforces that ZKC and StakingRewards epochs are aligned.

Formal Specification

```
vars: StakingRewards rewards, uint256 epoch, address user, ZKC zkc
inv: (
    # past reward values should never change
    (rewards.getCurrentEpoch() = 0)
    ||
    (
        epoch < old(rewards.getCurrentEpoch())
        && old(rewards.calculateRewards(user, [epoch]))[0] = rewards.calculateRewards
    (user, [epoch])[0]
    )
    ||
    # current/future rewards should always be zero
    (
        epoch >= old(rewards.getCurrentEpoch())
        && old(rewards.calculateRewards(user, [epoch])[0] = 0)
    )
# ZKC epoch matches reward epoch
) && zkc.getCurrentEpoch() = rewards.getCurrentEpoch()
```

6.3.19 V-ZKC-SPEC-019: Staking updates state correctly

Minutes Fuzzed	670	Bugs Found	1

Natural Language Description Staking creates a new active position, sets the staked amount to `amount`, leaves withdrawal time at 0, and reduces the sender's ZKC balance by `amount`. Preconditions include no prior active position, not withdrawing, sufficient ZKC balance, and `amount > 0`.

Formal Specification

```
vars: veZKC vez, ZKC zkc
# 2. Staking reduces ZKC balance by amount
inv: (
    # Staking implies {not withdrawing, no active position, have ZKC balance, non-zero stake amount}
    # and performing the stake withdraws the staked amount from the sender
    old(vez.getStakedAmountAndWithdrawalTime(sender)[0]) = 0
    && vez.getStakedAmountAndWithdrawalTime(sender)[0] = amount
    && old(vez.getStakedAmountAndWithdrawalTime(sender)[1]) = 0
    && vez.getStakedAmountAndWithdrawalTime(sender)[1] = 0
    && old(vez.getActiveTokenId(sender)) = 0
    && vez.getActiveTokenId(sender) != 0
    && old(zkc.balanceOf(sender)) >= amount
    && amount > 0
    && zkc.balanceOf(sender) = old(zkc.balanceOf(sender)) - amount
) over vez.stake(amount)
```

6.3.20 V-ZKC-SPEC-020: Upping stake by token ID updates state correctly

Minutes Fuzzed	670	Bugs Found	0

Natural Language Description Adds to stake for an explicit `tokenId`. Requires a valid, unchanged owner and existing stake with no withdrawal. It increases the owner's staked amount by `amount` and keeps withdrawal time at 0; the spec omits some assertions (active token id equality and ZKC balance change) due to performance timeouts.

Formal Specification

```
vars: veZKC vez, ZKC zkc
# 2. Staking reduces ZKC balance by amount
inv: (
    # Adding to stake implies {not withdrawing, has active position, have ZKC balance, non-zero
    # existing stake amount, and non-zero addition to stake}
    #
    # active token ID should not change
    tokenId != 0
    && old(vez.ownerOf(tokenId)) != 0
    && old(vez.ownerOf(tokenId)) = vez.ownerOf(tokenId)
    && old(vez.getStakedAmountAndWithdrawalTime(vez.ownerOf(tokenId))[0] > 0)
    && old(vez.getStakedAmountAndWithdrawalTime(vez.ownerOf(tokenId))[1] = 0)
    && (vez.getStakedAmountAndWithdrawalTime(vez.ownerOf(tokenId))[0] = old(vez.getStakedAmountAndWithdrawalTime(vez.ownerOf(tokenId))[0]) + amount)
    && (vez.getStakedAmountAndWithdrawalTime(vez.ownerOf(tokenId))[1] = 0)
    && (old(zkc.balanceOf(sender)) >= amount)
    && (amount > 0)
    # For some reason these cause timeouts
    # && old(vez.getActiveTokenId(vez.ownerOf(tokenId))) = tokenId
    # && vez.getActiveTokenId(vez.ownerOf(tokenId)) = tokenId
    # && (zkc.balanceOf(sender) = old(zkc.balanceOf(sender)) - amount)
) over vez.addToStakeByTokenId(tokenId, amount)
```

6.3.21 V-ZKC-SPEC-021: Upping stake updates state correctly

Minutes Fuzzed	730	Bugs Found	1

Natural Language Description Adding to stake requires an existing active position and no ongoing withdrawal. It keeps the same active token ID, increases the staked amount by `amount`, preserves withdrawal time at 0, and reduces the sender's ZKC balance by `amount`.

Formal Specification

```
vars: veZKC vez, ZKC zkc
# 2. Staking reduces ZKC balance by amount
inv: (
    # Adding to stake implies {not withdrawing, has active position, have ZKC balance, non-zero
    # existing stake amount, and non-zero addition to stake}
    #
    # active token ID should not change
    old(vez.getStakedAmountAndWithdrawalTime(sender)[0] > 0)
    && old(vez.getStakedAmountAndWithdrawalTime(sender)[1] = 0)
    && vez.getStakedAmountAndWithdrawalTime(sender)[0] = old(vez.getStakedAmountAndWithdrawalTime(sender)[0]) + amount
    && vez.getStakedAmountAndWithdrawalTime(sender)[1] = 0
    && old(vez.getActiveTokenId(sender)) != 0
    && old(vez.getActiveTokenId(sender)) = vez.getActiveTokenId(sender)
    && old(zkc.balanceOf(sender)) >= amount
    && amount > 0
    && zkc.balanceOf(sender) = old(zkc.balanceOf(sender)) - amount
) over vez.addToStake(amount)
```

6.3.22 V-ZKC-SPEC-022: Vote delegation changes state correctly

Minutes Fuzzed	670	Bugs Found	0

Natural Language Description When changing the voting delegate to a different address, the old delegate's voting power decreases by the user's staked amount, and the new delegate's voting power increases by the same amount.

Formal Specification

```
vars: veZKC vez, ZKC zkc, address oldDelegate
inv: (
    # Old delegate power decreases if the new one is different
    ((old(vez.delegates(sender)) != newDelegate) ==>
            vez.getVotes(old(vez.delegates(sender))) = old(vez.getVotes(old(vez.
    delegates(sender)))) - old(vez.getStakedAmountAndWithdrawalTime(sender)[0]))
    # New delegate power increseases if different from old one
    && ((old(vez.delegates(sender)) != newDelegate) ==>
            vez.getVotes(newDelegate) = old(vez.getVotes(newDelegate)) + old(vez.
    getStakedAmountAndWithdrawalTime(sender)[0]))
) over vez.delegate(newDelegate)
```

6.3.23 V-ZKC-SPEC-023: Vote delegation meets preconditions

Minutes Fuzzed	670	Bugs Found	0

Natural Language Description Changing the voting delegate requires an active position and preserves both the active token ID and staked amount. It also requires that no withdrawal is in progress (withdrawal time remains 0).

Formal Specification

```
vars: veZKC vez, ZKC zkc, address oldDelegate
inv: (
    # Must have an active position
    old(vez.getActiveTokenId(sender)) != 0
    # Token ID doesn't change
    && old(vez.getActiveTokenId(sender)) = vez.getActiveTokenId(sender)
    # Must have amount staked (implied by having an active position)
    && old(vez.getStakedAmountAndWithdrawalTime(sender)[0]) > 0
    # Staked amount doesn't change
    && vez.getStakedAmountAndWithdrawalTime(sender)[0] = old(vez.
    getStakedAmountAndWithdrawalTime(sender)[0])
    # Cannot be withdrawing
    && old(vez.getStakedAmountAndWithdrawalTime(sender)[1]) = 0
    && vez.getStakedAmountAndWithdrawalTime(sender)[1] = 0
) over vez.delegate(newDelegate)
```

6.3.24 V-ZKC-SPEC-024: ZKC Emission rate is respected

Minutes Fuzzed	790	Bugs Found	1

Natural Language Description The claimed total supply never exceeds the on-chain total supply, and the total supply is capped by a time-based emission schedule: before each scheduled epoch start, supply stays under the corresponding threshold.

Formal Specification

```
vars: ZKC zkc
inv: (
    # never have more tokens in circulation than expected total supply
    zkc.claimedTotalSupply() <= zkc.totalSupply()
    && (
        (timestamp >= zkc.getEpochStartTime(0) || zkc.totalSupply() <=
    100000000000000000000000000)
        && (timestamp >= zkc.getEpochStartTime(1 * 182) || zkc.totalSupply() <=
    106999999999999981840000000000)
        && (timestamp >= zkc.getEpochStartTime(2 * 182) || zkc.totalSupply() <=
    113954999999999537640000000000)
        && (timestamp >= zkc.getEpochStartTime(3 * 182) || zkc.totalSupply() <=
    120792299999999680269850000000)
        && (timestamp >= zkc.getEpochStartTime(4 * 182) || zkc.totalSupply() <=
    127435876499999091093195449750000)
        && (timestamp >= zkc.getEpochStartTime(5 * 182) || zkc.totalSupply() <=
    133807670324998831068124722700000)
        && (timestamp >= zkc.getEpochStartTime(6 * 182) || zkc.totalSupply() <=
    139829015489623516470771838800000)
        && (timestamp >= zkc.getEpochStartTime(7 * 182) || zkc.totalSupply() <=
    145422176109208245288644245500000)
        && (timestamp >= zkc.getEpochStartTime(8 * 182) || zkc.totalSupply() <=
    150511952273030237212507531300000)
    )
)
```

A Appendix

A.1 Intended Behavior: Non-Issues of Note

A.1.1 V-ZKC-INFO-001: Epoch times incorrect before 'initializeV2()'

Severity	Warning	**Commit**	f6f8f89
Type	Logic Error	**Status**	Intended Behavior
Location(s)	`src/ZKC.sol`		

As shown in the below snippet, the `epoch0StartTime` is not set until `initializerV2()` is called. This means that, prior to invoking `initializeV2()`, the `epoch0StartTime` is zero.

```
/// @dev On upgrade, set the epoch 0 start time to initiate the start of the first
    epoch.
function initializeV2() public reinitializer(2) {
    epoch0StartTime = block.timestamp;
}
```

However, contracts may still invoke `ZKC`-related functionality during this time. While `initializeV2()` has not been called, the `getCurrentEpoch()` function will return a very large epoch number.

```
function getCurrentEpoch() public view returns (uint256) {
    return (block.timestamp - epoch0StartTime) / EPOCH_DURATION;
}
```

Since the supply is larger in the future, this means that the emissions in the *reported* epoch will be much larger than the intended initial `ZKC` emissions. For example, if this time period overlapped with an epoch boundary, a user who had already staked `ZKC` could claim rewards for the epoch `(block.timestamp - 0) / EPOCH_DURATION` to obtain inflated rewards. More concerning, if an attacker could set the `PovwAccounting` contract to be stuck on an epoch which will not terminate for multiple decades.

```
/// Finalize the pending epoch, logging the finalized epoch number and total work.
function finalizeEpoch() public {
    uint256 newEpoch = TOKEN.getCurrentEpoch();
    require(pendingEpoch.number < newEpoch, "pending epoch has not ended");

    _finalizePendingEpoch(newEpoch);
}

/// End the pending epoch and start the new epoch. This function should
/// only be called after checking that the pending epoch has ended.
function _finalizePendingEpoch(uint256 newEpoch) internal {
    // Emit the epoch finalized event, accessed with Steel to construct the mint
    authorization.
```

```
13      emit EpochFinalized(uint256(pendingEpoch.number), uint256(pendingEpoch.totalWork)
        );
14
15      // NOTE: This may cause the epoch number to increase by more than 1, if no
        updates occurred in
16      // an interim epoch. Any interim epoch that was skipped will have no work
        associated with it.
17      pendingEpoch = PendingEpoch({number: newEpoch, totalWork: 0});
18  }
```

Snippet A.1: Snippet from `PovwAccounting`. If `finalizeEpoch()` is called while `epoch0StartTime` is zero, then the pending epoch will be set to a value roughly 55 years in the future. Once `initializeV2()` is called, the epoch will not be finalizable until those 55 years have passed.

Impact Attackers may DoS the rewards from `PovwAccounting` by setting it to an epoch far in the future.

Rewards interactions taken before the second initialization of the protocol may violate the ZKC emission schedule.

At the time of writing this issue (UNIX timestamp 1756314325), an epoch 0 start time of zero causes the current epoch to be interpreted as roughly 55 years after the protocol start. As shown in the below python script, the per-epoch emissions at this point are roughly 6 times their expected value.

```
INITIAL_SUPPLY = 1_000_000_000 # ; // 1 billion ZKC
Y0_R_PER_EPOCH = 1000371819923688085 / 1e18 # ; // Year 0: 7.000% annual
SUPPLY_YEAR_9 = 155027310841220836080404502O / 1e18
FINAL_R_PER_EPOCH = 1000162424190707866 / 1e18 # // 3.000% annual (minimum)

sanity check

tol = 1e-13
assert abs(Y0_R_PER_EPOCH ** 182 - 1.07) < tol
assert abs(FINAL_R_PER_EPOCH ** 182 - 1.03) < tol

epoch emissions at start of year 0

epoch_emission_year_0 = Y0_R_PER_EPOCH * INITIAL_SUPPLY

epoch emissions at year 55

n = 55
assert n >= 9
supply_year_n = 1.03 ** (n - 9) * SUPPLY_YEAR_9
epoch_emission_year_n = FINAL_R_PER_EPOCH * supply_year_n

print(f"Emissions at Year  0: {round(epoch_emission_year_0)} / epoch")
print(f"Emissions at Year {n}: {round(epoch_emission_year_n)} / epoch")
# Emissions at Year  0: 1000371820 / epoch
# Emissions at Year 55: 6039362310 / epoch
```

Recommendation Initialize the `epoch0StartTime` to be far in the future, preventing emissions until `initializeV2()` is called.

See also Non-atomic initialization.

Developer Response The developers indicated they have already deployed V1, and are planning to upgrade to V2 atomically.

Glossary

ERC-20 The famous Ethereum fungible token standard. See https://eips.ethereum.org/EIPS/eip-20 to learn more. 1

ERC-721 The Ethereum non-fungible token standard. See https://eips.ethereum.org/EIPS/eip-721 to learn more. 1

zero-knowledge circuit A cryptographic construct that allows a prover to demonstrate to a verifier that a certain statement is true, without revealing any specific information about the statement itself. See https://en.wikipedia.org/wiki/Zero-knowledge_proof for more. 54

zkVM A general-purpose zero-knowledge circuit that implements proving the execution of a virtual machine. This enables general purpose programs to prove their execution to outside observers, without the manual constraint writing usually associated with zero-knowledge circuit development . 1