OpenZeppelin | security

Boundless PoVW Audit

September 4, 2025

Table of Contents

Summary

Type	Blockchain Infrastructure
Timeline	From 2025-08-21 To 2025-08-26
Languages	Rust Solidity

Total Issues	21 (3 resolved)
Critical Severity Issues	0 (0 resolved)
High Severity Issues	0 (0 resolved)
Medium Severity Issues	3 (2 resolved)
Low Severity Issues	5 (1 resolved)
Notes & Additional Information	13 (0 resolved)

Scope

OpenZeppelin audited the boundless-xyz/boundless repository at commit 8bd70a8.

In scope were the following files:

```
contracts/
└── src/
    └── povw/
        └── PovwMint.sol
crates/
└── guest/
    └── povw/
        ├── mint-calculator/
        │   └── src/
        │       └── main.rs
        └── src/
            └── mint_calculator.rs
```

System Overview

Boundless is a universal zero-knowledge (ZK) protocol that extends verifiable computing across any blockchain. By abstracting away the complexities of proof generation, aggregation, and on-chain settlement, Boundless enables developers to focus on building high-throughput and expressive applications that bypass the traditional block-size and gas limits.

It introduces a permissionless, decentralized marketplace for proof generation, where developers submit proof requests and independent provers compete to fulfill them, earning direct rewards and protocol-level incentives through a Proof of Verifiable Work (PoVW) model. Apart from direct rewards paid by the developers, the Boundless protocol enables provers to additionally receive ZKC tokens, the amount of which depends on the total proving work done and the amount of staked ZKC tokens.

In order to account for the work being done, a prover submits on-chain an update to a "*work log*" they own, along with a PoVW demonstrating the validity and uniqueness of the claimed work. Each such submission results in an emission of the `WorkLogUpdate` event, confirming it and including the `initialCommit` and `updatedCommit` values, which correspond to the roots of the Merkle trees encompassing all the PoVW proofs submitted before and right after an update.

Submissions are made during *epochs*, which are currently set to last several days. Provers are rewarded based on the following three factors:

- An epoch's total token emissions
- The percentage of contribution to the total work performed in the epoch
- The reward cap, which depends on the amount of staked ZKC tokens

Newly Introduced Logic

The code under review implements logic for the calculation of ZKC token rewards described above and consists of three different components:

- The guest program, executing on zkVM
- The host program providing input for the guest
- The on-chain component

Guest Program

The main task of the guest program is to process `WorkLogUpdate` events from the set of specified blocks for the selected work logs. The result of the execution of the guest program is an input for the on-chain contract (described below) along with the proof of the performed computation.

The input for the on-chain contract consists of two lists:

- A list of work log commitment updates
- A list of recipients and the amounts of tokens each of them should receive

Both of these lists are populated based on the events emitted by the `PovwAccounting` contract and the results of `view` calls performed on the state of the `ZKC` and `ZKCRewards` contracts at the end of the processed blocks.

While constructing these lists, the guest program ensures that:

- work log updates have been processed in the correct order (that is, the `initialCommit` of an update is equal to the `updatedCommit` of a previous update)
- no work log update has been skipped
- the list of updates for each work log returned at the end of the computation is a compressed list, only containing the `initialCommit` of the first processed update and the `updatedCommit` of the last processed update
- if a work log update has been processed for a given epoch, then all updates to that work log have been also processed for that epoch
- reward caps are imposed on mint recipients based on the data acquired on-chain at the end of each processed epoch
- processed blocks are cryptographically bound to each other

It is important to note that the input supplied to the guest program may be arbitrary. Hence, the guest program has to verify the data that it received. This is achieved by using the Steel framework, which enables accessing data from EVM blockchains and verifying that it is consistent with the provided block headers.

Host Program

The main task of the host program is to fetch on-chain data and pass it as an input to the guest program. This is necessary as the guest program executes inside the zkVM and, thus, cannot directly query on-chain data.

In order to construct the input for the guest program, the host program queries an RPC provider for the list of specified blocks and fetches all the data which will be queried by the guest program from these blocks. This is achieved by utilizing the `preflight` feature of the Steel framework, which allows for creating an EVM environment exposing only the requested part of data.

On-Chain Component

The on-chain component is the `PovwMint` smart contract, which records updates to each provided work log by saving the `updatedCommit` value from the compressed list of updates provided by the guest program and mints tokens to the specified recipients. In order to make sure that the data has been provided by a trusted guest program, `PovwMint` requires the correct guest image ID, encompassing the guest's compiled code, to be included in the provided ZK proof, which it also verifies.

Furthermore, since the guest program cannot fully guarantee the correctness of the data it processes, the `PovwMint` contract performs the following additional verifications:

- All the contract addresses from which the guest program queried the data are correct
- The last block processed from the guest does indeed belong to the chain where the `PovwMint` contract is deployed
- The last recorded commit for a given work log matches the `initialCommit` from the compressed list of updates provided by the guest program

These verifications, combined with the checks performed by the trusted guest program, guarantee that the correct data has been supplied and that no updates have been skipped.

Security Model and Trust Assumptions

Due to the number of out-of-scope components involved and the extensive usage of the Steel framework, multiple trust assumptions were made during the audit:

- All out-of-scope components work correctly, which include, but are not limited to, the `PovwAccounting`, `RiscZeroVerifier`, `ZKC`, and `ZKCRewards` contracts. In particular, it is trusted that the `PovwAccounting` contract correctly verifies submitted proofs of the performed work, correctly updates the total work recorded for an epoch,

and guarantees that the `updatedCommit`s recorded in the `WorkLogUpdate` events are unique per each update.
- The relevant events are emitted upon each epoch's end.
- The `RiscZeroVerifier` contract correctly verifies the provided ZK proofs.
- The `ZKC` and `ZKCRewards` contracts store and return the expected and correct values, such as past reward caps or epoch end times, which are later queried during the calls performed by the guest program.
- It is assumed that the logic related to the reward cap per epoch in the out-of-scope components is implemented in the correct and reasonable way, not allowing any manipulations.
- It is assumed that the `PovwMint` contract will be configured correctly, with all the parameters set to the correct values.
- It is assumed that the Steel framework behaves in the expected correct way. This includes verifying the integrity of the data against the supplied block headers when the guest program queries it and correctly exposing the data which has been prepared by the host through the `preflight` feature.
- The exposed data, such as emitted events, are ordered by their order of emission in each block.

Medium Severity

M-01 Reward Cap Can Be Bypassed

The Mint Calculator guest program imposes a cap on the rewards minted for each recipient. In order to ensure that the work log updates in an epoch cannot be split to bypass this cap, it is required that all epochs containing them are fully processed. However, this cap could still be bypassed by a prover who could own multiple work logs and specify a common reward recipient for all of them. This way, the reward cap will be imposed on each work log separately and, as a consequence, the total amount of minted tokens through different work logs in the same epoch can exceed the cap.

Consider not allowing to specify a custom mint recipient and instead minting tokens to the `workLogId` address. Alternatively, consider storing on-chain the total amount of rewards minted in each epoch for each recipient and imposing the reward cap taking into account this amount.

Update: *Resolved in pull request #1038.*

M-02 Rewards May Be Temporarily Frozen

When the Mint Calculator guest program processes the update events, it ensures that all events from all parsed epochs have been fully processed for each referenced work log. This is to ensure that the reward cap for each processed epoch cannot be bypassed by splitting the work for a single epoch into multiple chunks.

However, if a `WorkLogUpdated` event for a certain work log has been emitted in the same block as the last `EpochFinalized` event, the last `updated_commit` stored in the `updates` map will be different from the final commit accessed in the previous block, hence the completeness check will not pass in such a case. This will block any token mints for that work log until the next epoch, assuming that no update for that work log happens in the same block as the next `EpochFinalized` event.

During event-processing, consider not taking into account the `WorkLogUpdated` events emitted in the same block or future blocks from the last `EpochFinalized` event.

Update: *Resolved in pull request #1039.*

M-03 `build` Function for Beacon Commitments Will Panic

In the Mint Calculator host program, there are two, almost identical implementations of the `build` function for different commitment types. A notable difference includes different behavior on an attempt to insert elements into the `multiblock_env.0 BTreeMap` - the first implementation ignores the value returned by `insert(...)`, while the second one unwraps the returned value, which will cause panic on an attempt to insert the first element to that map. As a result, this function will not be able to process any blocks.

Consider not unwrapping the result returned by the `multiblock_env.0.insert(...)` call in order to avoid panics in the host program.

Update: *Acknowledged, will resolve. The RISC Zero team stated:*

> *This feature was completed after the reference audit commit as part of ongoing feature work. The code in question has no impact on the soundness of the reward mechanism.*

Low Severity

L-01 Usage of Unstable API

The `main` function of the Mint Calculator guest program uses the `read_frame` function to get the input data from the host program. However, the `read_frame` function is marked as unstable in the `risc0_zkvm` crate, which means that it is subject to change or removal in the future.

Consider using a stable alternative for receiving data inside the guest program, such as the `read` function. Alternatively, consider documenting the usage of the unstable API so that the intention and potential risks involved when using it are well communicated.

Update: *Acknowledged, not resolved. The RISC Zero team stated:*

> *The `risc0-zkvm` crate is maintained by our team. We have marked this API as unstable to indicate that we may remove it in the future. If we do, this code will be updated.*

L-02 Prover Could Lose Future Rewards Upon Specifying `address(0)` as the Recipient

The `mint` function of the `PovwMint` contract mints given amounts of tokens to the specified recipients. Before the actual minting, it performs the verification that all the data has been accounted for and that no work log update was skipped.

However, the `mintPoVWRewardsForRecipient` call reverts when the `mintData.recipient` is specified as the `address(0)`, as the actual implementation of the `ZKC` token relies on the OpenZeppelin implementation, which rejects mints to `address(0)`. As a result, if a reward cap for `address(0)` is a non-zero value for a given epoch, a prover who mistakenly specifies `address(0)` as the recipient in a `PovwAccounting.updateWorkLog` call does not only lose access to the reward for a single work log update, but loses the rewards for all subsequent updates since it is not possible to skip any update.

Consider explicitly handling the case of `mintData.recipient` being equal to `address(0)` inside the `PovwMint.mint` function and not reverting in such a case. Alternatively, if it is not possible for `address(0)` to have a non-zero reward cap, consider documenting this fact in the code.

Update: *Resolved in pull request #1055.*

L-03 `PovwMint.mint` Function Can Run Out of Gas

At the end of the `mint` function of the `PovwMint` contract, token rewards are minted to the specified recipients in a loop. However, since reward recipients do not have to be equal to work log owners, it is possible that, for a single work log and epoch, the number of recipients is so big that the loop can exceed the block gas limit during the execution on Ethereum. As a result, it would not be possible to mint the rewards for a given epoch and all subsequent epochs for that work log.

Consider documenting this edge case in the code, warning work log owners from specifying too many different reward recipients in a single epoch.

Update: *Acknowledged, not resolved. The RISC Zero team stated:*

> *The provided tooling for submitting log updates is designed to submit at most one log update per block. Triggering this condition would require submitting hundreds of log*

> *updates in a single block. This would require custom tooling and would be very expensive in terms of gas. It is unclear if there is any benefit to doing so, whereas if this condition is triggered, only that prover would lose their rewards.*

L-04 Variables in the `PovwMint` Contract Could Be Made `public`

The `PovwMint` contract contains several `internal` variables, which are either set in the `constructor` or updated in the `mint` function. However, since these variables are not declared as `public`, there is no straightforward way to read their values.

Consider making the `internal` variables inside the `PovwMint` contract `public` so that they are easily accessible both on-chain and off-chain.

Update: *Acknowledged, will resolve.*

L-05 Lack of Event Emissions

The `constructor` and the `mint` function of the `PovwMint` contract update `internal` variables specified in that contract. However, neither of these functions emits any event upon performing these changes. As such, it is harder to keep track of the updates made to the `latestCommit` mapping, which is essential in order to be able to provide correct parameters to the `mint` function that would otherwise revert. Similarly, without events emitted in the `constructor`, it is harder to determine the initial values assigned to the `immutable` variables.

Consider emitting relevant events on changes made to the global variables of the `PovwMint` contract in order to make the process of determining the right input for the `mint` function easier.

Update: *Acknowledged, will resolve.*

Notes & Additional Information

N-01 `MINT_CALCULATOR_ID` Could Be Mutable

The `MINT_CALCULATOR_ID` variable in the `PovwMint` contract represents the image ID of the Mint Calculator guest program. This image ID is then used for the verification of ZK proofs provided to the `PovwMint.mint` function. However, in case there is a necessity to change any logic in the Mint Calculator guest implementation, and by doing that, change its image ID, it will only be possible through redeploying the `PovwMint` contract.

Consider making the `MINT_CALCULATOR_ID` variable mutable in order to allow the modifications of the Mint Calculator guest program in the `PovwMint` contract.

Update: *Acknowledged, will resolve.*

N-02 Typographical Errors

Throughout the codebase, multiple instances of typographical errors have been identified:

- In line 31 of the `PovwMint.sol` file, "collasped" should be "collapsed".
- In line 79 of the `PovwMint.sol` file, "ensure" should be "ensures".
- In line 36 of the `mint-calculator/src/main.rs` file, "from" should be "into".
- In line 40 of the `mint-calculator/src/main.rs` file, "a" is unnecessary and could be removed.
- In line 121 of the `mint-calculator/src/main.rs` file, "to avoid" could be removed.
- In line 130 of the `mint-calculator/src/main.rs` file, the `completness_check_env` should be "`completeness_check_env`".
- In line 82 of the `src/mint_calculator.rs` file, "which indirectly commitment" should be "which is indirectly a commitment".
- In line 321 of the `src/mint_calculator.rs` file, "name" should be "same".
- In line 468 of the `src/mint_calculator.rs` file, "lastest_env" should be "latest_env".

Consider correcting the above-listed instances of typographical errors in order to improve the clarity and readability of the codebase.

Update: *Acknowledged, will resolve.*

N-03 Redundant Loop

In order to determine the latest block where the `EpochFinalized` event has occurred, the Mint Calculator guest program fetches all `EpochFinalized` events from each processed block and updates the `latest_epoch_finalization_block` variable in a loop.

However, as pointed out by this comment, the loop can iterate at most once per each `EpochFinalized` event query as only one such event can be emitted in a block by the `PovwAccounting` contract. As such, in order to avoid confusion and simplify the code, the loop could be replaced with an `if let` statement performed on the first element of the `epoch_finalized_events` vector.

Consider replacing the loop over the `epoch_finalized_events` vector with a relevant `if let` statement in order to increase code clarity.

Update: *Acknowledged, will resolve.*

N-04 Incorrect and Unused `IPovwMint` Interface

The `IPovwMint` interface declared in the Mint Calculator host program contains the signatures of the `mint` and `lastCommit` functions. However, the `lastCommit` function is not present in the `PovwMint` contract. While there is a mapping with a similar name in that contract, it has been declared as `internal` and, as a result, the Solidity compiler does not generate a getter function for it. Furthermore, the `IPovwMint` interface is not used anywhere in the code.

Consider removing the incorrect function signature from the `IPovwMint` interface or changing the visibility of the `latestCommit` mapping to `public` inside the `PovwMint` contract and specifying the correct signature for its getter function in the interface. Alternatively, if the `IPovwMint` interface is not intended to be used, consider removing it.

Update: *Acknowledged, will resolve.*

N-05 Misleading or Missing Comments

Throughout the codebase, multiple instances of misleading comments have been identified:

- This comment refers to the code three lines below, while this comment refers to the line directly above.
- This comment suggests that if `work_log_filter` is not specified, then all work logs with updates in the given blocks will be included, and if it is specified, then only a given set of work log IDs will be included. However, `work_log_filter` is always specified in the `Input` struct and only the internal `BTreeSet` could potentially be unspecified to imply that all work log IDs should be processed. In addition, the meaning of the keys and values of the updates mapping are not documented.

In order to improve the clarity and maintainability of the codebase, consider addressing the aforementioned instances of misleading documentation, as well as including any missing documentation.

Update: *Acknowledged, will resolve.*

N-06 Implicit Block Inclusion Requirement

Both Mint Calculator guest and host programs iterate over the received set of blocks and require that a block which is a direct predecessor of the last block containing the `EpochFinalized` event is included in that set [1] [2]. This is needed to ensure that all `WorkLogUpdated` events have been fully processed for the last epoch. However, the requirement of inclusion of that block is not obvious without looking at the source code, especially due to the fact that it does not necessarily contain any `EpochFinalized` or `WorkLogUpdated` event.

Consider documenting the necessity of including the block which was immediately before the block containing the last `EpochFinalized` in the input for the `Input::build` function.

Update: *Acknowledged, will resolve.*

N-07 Unaddressed TODO Comments

During development, having well-described TODO comments will make the process of tracking and solving them easier. Without this information, these comments might age and important information for the security of the system might be forgotten by the time it is released to

production. As such, these comments should be tracked in the project's issue backlog and resolved before the system is deployed.

Throughout the codebase, multiple instances of TODO comments have been identified.

Consider removing all instances of TODO comments and instead tracking them in the issues backlog. Alternatively, consider linking each inline TODO comment to the corresponding issues backlog entry.

Update: *Acknowledged, will resolve.*

N-08 Custom Errors in `require` Statements

Since Solidity version `0.8.26`, custom error support has been added to `require` statements. Initially, this feature was only available through the IR pipeline. However, Solidity `0.8.27` extended its support to the legacy pipeline as well.

The `PovwMint` contract contains multiple instances of `if-revert` statements that could be replaced with `require` statements with custom errors.

Consider replacing `if-revert` statements with `require` ones for conciseness.

Update: *Acknowledged, will resolve.*

N-09 Missing Security Contact

Providing a specific security contact (such as an email address or ENS name) within a smart contract significantly simplifies the process for individuals to communicate if they identify a vulnerability in the code. This practice is quite beneficial as it permits the code owners to dictate the communication channel for vulnerability disclosure, eliminating the risk of miscommunication or failure to report due to a lack of knowledge on how to do so. In addition, if the contract incorporates third-party libraries and a bug surfaces in those, it becomes easier for their maintainers to contact the appropriate person about the problem and provide mitigation instructions.

The `PovwMint.sol` file does not have a security contact.

Consider adding a NatSpec comment containing a security contact above each contract definition. Using the `@custom:security-contact` convention is recommended as it has been adopted by the OpenZeppelin Wizard and the ethereum-lists.

Update: *Acknowledged, will resolve.*

N-10 Missing SPDX License Identifier

The `PovwMint.sol` file is missing an SPDX license identifier.

In order to follow best practices, consider adding SPDX license identifiers to files as suggested by the Solidity documentation.

Update: *Acknowledged, will resolve.*

N-11 Input Validation Can Happen Earlier

In line 103 of the Mint Calculator guest program, if the epoch number of any `WorkLogUpdated` event that has not been filtered out by the work log filter is not already present in the `epochs` map, the program will crash. However, the epoch-number validation can be performed earlier in order to avoid executing unnecessary operations in case the referenced epoch is not present in the `epochs` map.

Consider performing the epoch number validation before the processing of the `WorkLogUpdated` event in order to stop the proof generation earlier in case incorrect data has been supplied.

Update: *Acknowledged, will resolve.*

N-12 Code Can Be Simplified by Currying `rewards_weights`

A nested mapping `f: x -> (g: y -> z)` is logically equivalent to a mapping `h: (x, y) -> z` whose keys are ordered pairs. Indeed, this comment inside the Mint Calculator guest program refers to `rewards_weights` as a "mapping of calculated rewards, with the key as (epoch, recipient) pairs and the value as a `FixedPoint` fraction [...]". However, the `rewards_weights` mapping is constructed as a nested `BTreeMap` of the form `epoch -> (recipient -> reward fraction)`. This complicates the code for updating `rewards_weights`.

To simplify and potentially optimize the code, consider constructing `rewards_weights` as described in the documentation: as a mapping of the form `(epoch, recipient) ->`

`reward fraction`. Alternatively, consider amending the documentation to reflect that `rewards_weights` is a nested mapping.

Update: *Acknowledged, will resolve.*

N-13 Unused Import in `PovwMint` Contract

The `PovwMint` contract includes an import statement for the Math library from OpenZeppelin's contracts suite. However, the Math library is not utilized within the contract. While this unnecessary import does not impact the contract's functionality or security, it can potentially lead to confusion regarding the contract's dependencies.

Consider removing the redundant import for improved clarity and maintainability.

Update: *Acknowledged, will resolve.*

Conclusion

The Boundless protocol introduces a sophisticated mechanism for verifiable work accounting and reward distribution, leveraging zero-knowledge proofs and the Steel framework to ensure correctness and integrity across multiple components. The architecture demonstrates a thoughtful separation of responsibilities between the guest, host, and on-chain contracts, with robust verification steps at each stage to mitigate the risk of skipped updates or inconsistent data.

The codebase was found to be well structured and thoroughly documented, which greatly facilitated the review. Clear documentation and comprehensive comments provided valuable insights into the intended logic and security considerations, reflecting a strong commitment to code quality and maintainability.

The RISC Zero team is appreciated for their professionalism and responsiveness during the engagement. Their detailed protocol walkthrough and prompt responses to questions were instrumental in deepening the audit team's understanding of the system, ensuring a thorough assessment.

Appendix

Issue Classification

OpenZeppelin classifies smart contract vulnerabilities on a 5-level scale:

- Critical
- High
- Medium
- Low
- Note/Information

Critical Severity

This classification is applied when the issue's impact is catastrophic, threatening extensive damage to the client's reputation and/or causing severe financial loss to the client or users. The likelihood of exploitation can be high, warranting a swift response. Critical issues typically involve significant risks such as the permanent loss or locking of a large volume of users' sensitive assets or the failure of core system functionalities without viable mitigations. These issues demand immediate attention due to their potential to compromise system integrity or user trust significantly.

High Severity

These issues are characterized by the potential to substantially impact the client's reputation and/or result in considerable financial losses. The likelihood of exploitation is significant, warranting a swift response. Such issues might include temporary loss or locking of a significant number of users' sensitive assets or disruptions to critical system functionalities, albeit with potential, yet limited, mitigations available. The emphasis is on the significant but not always catastrophic effects on system operation or asset security, necessitating prompt and effective remediation.

Medium Severity

Issues classified as being of medium severity can lead to a noticeable negative impact on the client's reputation and/or moderate financial losses. Such issues, if left unattended, have a moderate likelihood of being exploited or may cause unwanted side effects in the system.

These issues are typically confined to a smaller subset of users' sensitive assets or might involve deviations from the specified system design that, while not directly financial in nature, compromise system integrity or user experience. The focus here is on issues that pose a real but contained risk, warranting timely attention to prevent escalation.

Low Severity

Low-severity issues are those that have a low impact on the client's operations and/or reputation. These issues may represent minor risks or inefficiencies to the client's specific business model. They are identified as areas for improvement that, while not urgent, could enhance the security and quality of the codebase if addressed.

Notes & Additional Information Severity

This category is reserved for issues that, despite having a minimal impact, are still important to resolve. Addressing these issues contributes to the overall security posture and code quality improvement but does not require immediate action. It reflects a commitment to maintaining high standards and continuous improvement, even in areas that do not pose immediate risks.