Pod Network Bridge Audit Report

Prepared by Riley Holterhus

January 29, 2026

Contents

1 Introduction

1.1 About Pod Network

Pod Network is a Layer-1 network with an architecture that differs from most traditional blockchains. One notable aspect is that transactions are partially ordered rather than enforced under a strict total order, which allows for faster transaction processing.

1.2 About the Auditor

Riley Holterhus is an independent security researcher who focuses on Solidity smart contracts. Other than conducting independent security reviews, he works as a Lead Security Researcher at Spearbit, and also searches for vulnerabilities in live codebases. Riley can be reached by email at rileyholterhus@gmail.com, by Telegram at @holterhus and on Twitter/X at @rileyholterhus.

1.3 Disclaimer

2 Audit Overview

2.1 Scope of Work

From January 24, 2026 to January 25, 2026, Riley Holterhus conducted a manual security review of Pod's `Bridge` smart contract. The goal was to identify potential vulnerabilities and logic issues in the codebase.

The audit was performed on the codebase found in the `podnetwork/pod-sdk` GitHub repository. The audit started on commit 078fbab. The files in scope for the audit were:

- `protocol/src/Bridge.sol`
- `protocol/src/lib/ProofLib.sol`

The `Bridge` contract is intended to be deployed on an EVM chain such as Ethereum mainnet or Base, and is responsible for bridging tokens between the EVM chain and Pod. Users deposit ERC20 tokens into the contract, which is observed by a validator set and results in corresponding tokens being minted on Pod. When bridging tokens back from Pod, users receive a set of signatures (or an equivalent proof against an admin-provided merkle tree) that can be submitted to the contract to transfer tokens out. The contract also includes migration logic to allow the admin to move funds into a new contract in the event of an issue.

2.2 Summary of Findings

Each finding from the audit has been assigned a severity level of “Critical”, “High”, “Medium”, “Low” or “Informational”. These severities aim to capture the impact and likelihood of each potential issue.

In total, **7 findings** were identified: 1 Medium, 2 Low, and 4 Informational. The description and status of the findings are provided in the sections below.

3 Findings

3.1 Medium Severity Findings

3.1.1 Version updates require careful merkle tree construction

Description: The `Bridge` allows the admin to update the `version` number, which updates the `domainSeparator` and causes all subsequent `txHash` calculations to be derived using the new `version`. There is a comment above `updateValidatorConfig()` that references this behavior:

```
/**
 * ...
 * @dev Admin may keep the same version if the config update doesn't invalidate past certificates.
 *      When the version is updated, past certificates become invalid and only merkle proofs work
 *      for claims from before the version change.
 * ...
 */
```

While this comment indicates that merkle proofs are intended to be the mechanism for proving unclaimed transactions from a previous `version`, the exact mechanics of how this is expected to work are more nuanced.

In order for merkle proofs to be used for transactions from a previous `version`, an updated merkle root would need to be submitted that includes all previously unclaimed transactions, with their leaves recomputed using the new domain separator. Since this effectively reintroduces the same transactions under new `txHash` values, the update process must be careful not to include any transactions that were already claimed under the old version, as they would become claimable again under the new `txHash` in the merkle tree.

So, for example, it's important that the `version` upgrade and the updating of the merkle root are not performed in a single step based on an assumed set of unclaimed transactions. If both were updated simultaneously, an attacker could front-run the update by claiming a transaction under the old version, and then claim the same transaction again after the update using the new merkle root. To avoid this race condition, the version update and merkle root update would need to be performed as separate steps. This approach could be error-prone and is currently not documented in the code.

Recommendation: Consider changing this behavior to be more robust. One possible approach would be to allow the claimer to specify the `version` associated with the transaction being claimed. This would allow the `txHash` for a given transaction to remain the same across `version` updates, which in turn would allow merkle root updates to include transactions that were already claimed, without introducing a risk of double claims.

Pod: Fixed in commit 6e7293b.

Auditor: Verified.

3.2 Low Severity Findings

3.2.1 Upgradeable contract inherits base `AccessControl`

Description: The `Bridge` contract is intended to be upgradeable behind a proxy, but it is inheriting OpenZeppelin's base `AccessControl` contract rather than `AccessControlUpgradeable` from the upgradeable OpenZeppelin codebase.

In practice, using `AccessControl` behind a proxy doesn't introduce any immediate issues, as it doesn't have any special constructor or initialization logic. However using `AccessControlUpgradeable` would be more consistent and also more forward-compatible, since the upgradeable version uses an ERC-7201 storage layout.

Recommendation: Consider inheriting the `AccessControlUpgradeable` contract instead of `AccessControl`.

Pod: Fixed in commit 8aeedab.

Auditor: Verified. The `AccessControlUpgradeable` contract is now being inherited.

3.2.2 `minAmount` and `claimLimit` checks can result in unclaimable funds

Description: Every token in the `Bridge` is configured with `minAmount`, `depositLimit`, and `claimLimit` thresholds.

For claims, these thresholds can theoretically create unclaimable situations. For example, if a user burns tokens on Pod to bridge back to mainnet with an amount below `minAmount`, or if `minAmount` is increased after the burn, the user wouldn't be able to claim their tokens through the `Bridge`. Another example is if a user bridges back an amount larger than the token's `claimLimit`. This is unclaimable because claims can't be split up into smaller pieces once they're initiated and signed.

Recommendation: Ensure users do not initiate bridge operations back to mainnet with amounts below `minAmount` or above the `claimLimit`, as this would result in stuck funds. If possible, enforce these thresholds at the time the bridge is initiated on the Pod side. Additionally, ensure that later increases to `minAmount` or decreases to `claimLimit` do not create stuck funds.

Pod: Acknowledged.

Auditor: Acknowledged.

3.3 Informational Findings

3.3.1 Invariant between `validatorCount` and `adversarialResilience` is split across functions

Description: The `_addRemoveValidators()` and `_setAdversarialResilience()` functions are admin helper functions. The `_addRemoveValidators()` function adds and removes validators from the active validator set, while `_setAdversarialResilience()` updates the `adversarialResilience` value, which must always be less than or equal to `validatorCount`.

Currently, `_addRemoveValidators()` does not verify that changes to the validator set preserve the invariant `validatorCount >= adversarialResilience`. Instead, this invariant is enforced indirectly by relying on a subsequent call to `_setAdversarialResilience()`. As a result, future changes to the code must ensure that `_addRemoveValidators()` is always followed by `_setAdversarialResilience()` to avoid violating this invariant.

Recommendation: Consider documenting this behavior for future versions of the code. Alternatively, consider combining `_addRemoveValidators()` and `_setAdversarialResilience()` into a single function so that the invariant is upheld more explicitly.

Pod: Fixed in commit a64ef9c.

Auditor: Verified.

3.3.2 Limit behavior is affected by order splitting

Description: The `Bridge` contract enforces both deposit and claim limits using a rolling usage tracker. Before each deposit or claim, the current usage is incremented by the requested amount and checked against the limit, as shown below:

```
uint256 newConsumed = usage.consumed + amount;
if (newConsumed > maxTotalAmount) {
    if (block.timestamp < usage.lastUpdated + 1 days || amount > maxTotalAmount) {
        revert DailyLimitExhausted();
    } else {
        usage.lastUpdated = block.timestamp;
        usage.consumed = amount;
    }
} else {
    usage.consumed = newConsumed;
}
```

Notice that if an action pushes the current usage past the limit, the usage is reset and the full amount is applied to the new period, causing any remaining capacity from the previous period to be lost. As a result, splitting an action into multiple smaller actions near a reset can lower the end usage, even when the overall amount is the same.

Recommendation: Consider changing the behavior so that splitting an action does not lead to a different end result. This can be achieved by filling remaining capacity first and only applying excess amounts to the usage in a new period.

Pod: Fixed in commit e864c76.

Auditor: Verified. The code now fills remaining capacity first and applies the excess to the new period.

3.3.3 `lenData` can be used instead of a hardcoded length

Description: In the `_depositTxHash()` function, `dataHash` is computed using a hardcoded length:

```
dataHash := keccak256(ptr, 0x64)
```

Note that this hardcoded `0x64` value already corresponds to the `lenData` variable defined in the function.

Recommendation: Consider replacing the hardcoded `0x64` value with the `lenData` variable.

Pod: Fixed in commit 9fa3752.

Auditor: Verified.

3.3.4 Usage resets favor faster claimers

Description: The `Bridge` enforces usage limits on both deposits and claims, which can be reset every 24 hours. After a reset, usage is allocated on a first-come first-served basis. This means that when usage resets, multiple users might rush to consume the new capacity.

This behavior could negatively affect the user-experience, especially for claims. For claims, bridging back to mainnet is a two-step process where users first burn their tokens on Pod and then submit their claim to the `Bridge` later. If the claim capacity is consistently consumed immediately after a reset, slower users may have to wait multiple days to claim, despite already having burned their tokens.

Recommendation: Keep this behavior in mind, and if it causes issues in practice, consider implementing a queue mechanism.

Pod: Acknowledged.

Auditor: Acknowledged.