OpenZeppelin | security

TxFlow Security Audit



July 15, 2026

This document has been prepared solely for the named client and it may not be relied upon by any third parties. The content in this document is provided for informational purposes only and does not constitute professional advice of any kind. Security assessments are limited by factors such as scope, time, and techniques used and therefore this document does not constitute a certification, guarantee, or warranty regarding the proper functioning of any system or the absence of security vulnerabilities.

Copyright © 2026 Zeppelin Group Ltd.

Table of Contents

Table of Contents 2

Summary 3

Scope 4

System Overview 5

Security Model and Trust Assumptions 5

Privileged Roles 7

Medium Severity 8

M-01 updateValidatorSet Can Be Replayed to Block Finalization 8

Low Severity 8

L-01 Irrecoverable DoS Via Malicious Validator Set Parameters 8

L-02 Missing Lower Bound Validation in Locker Threshold Update 9

L-03 Non-Standard EIP-712 Implementation 10

L-04 Deposits Possible While Bridge Is Paused 10

L-05 Permit Front-Running Can Temporarily Block Deposits 11

Notes & Additional Information 11

N-01 Custom Cryptographic Helpers Lack Standard Validation Safeguards 11

N-02 Naming Suggestions 12

N-03 Stale Data in pendingValidatorSetUpdate Variable After Finalization 13

N-04 Missing Event Emissions And Indexed Event Parameters Reduce Off-Chain Observability 13

N-05 Magic Numbers and String Literals Used in the Code 14

Conclusion 15

Appendix 16

Issue Classification 16

Summary

Type	DeFi
Timeline	From 2026-06-25 To 2026-06-30
Languages	Solidity

Total Issues	11 (5 resolved, 1 partially resolved)
Critical Severity Issues	0 (0 resolved)
High Severity Issues	0 (0 resolved)
Medium Severity Issues	1 (1 resolved)
Low Severity Issues	5 (4 resolved, 1 partially resolved)
Notes & Additional Information	5 (0 resolved)
Client Reported Issues	0 (0 resolved)

Scope

OpenZeppelin performed an audit of the `Bridge2` contract deployed at the `0xcde3f99bcb4c91e19124e41730489eaefec24565` address on Arbitrum.

In scope were the following files:

```
project/
└── contracts/
    ├── Bridge2.sol
    └── Signature.sol
```

System Overview

`Bridge2` is a custodial USDC bridge contract deployed on Arbitrum One that connects to the TxFlow L1. The bridge holds USDC on the Arbitrum side and relies on a validator set to authorize cross-chain operations. Each validator maintains a hot and cold wallet pair. Operations are authorized through EIP-712 structured messages signed by the validators.

The bridge supports two primary user flows. Withdrawals are initiated when L1 validators sign withdrawal requests, which are batched and submitted to the bridge. Each withdrawal enters a dispute period (verified against both wall-clock time via `block.timestamp` and Arbitrum block numbers via `ArbSys.arbBlockNumber()`) before a finalizer can release the USDC. Deposits are recognized by L1 validators monitoring USDC `Transfer` events to the bridge address and crediting users on L1. The bridge also enables gasless deposit approvals using EIP-2612 permits.

Validator set management follows an epoch-based lifecycle, with an expected epoch duration of 1-7 days. The active hot validators sign a hash of the incoming validator set, and the update enters a dispute period before finalization. The L1 consensus layer enforces slashing conditions to ensure outgoing validators cooperate with the transition.

The bridge includes an emergency locking mechanism in which designated lockers can vote to pause the contract. Once the number of votes reaches the configured `lockerThreshold`, the bridge pauses. Only a cold-quorum `emergencyUnlock` can restore operation, atomically rotating the validator set, clearing outstanding locker votes, and unpausing the bridge.

A custom EIP-712 implementation in `Signature.sol` handles domain separator construction, typed-data hashing, and ECDSA signature recovery.

Security Model and Trust Assumptions

The bridge follows a Byzantine fault-tolerant quorum model with a layered key architecture. Routine operations require a strict >2/3 hot-wallet quorum, while high-assurance actions

(unlocking, role removal, parameter changes) require a strict >2/3 cold-wallet quorum. A dispute period and locker-based pause mechanism provide a time-bound window for intervention before operations are finalized.

- **Validator majority honesty.** The bridge assumes that no more than 1/3 of the total validator power is controlled by malicious actors. A >2/3 hot-wallet quorum can propose arbitrary withdrawals and validator set updates, constrained only by the dispute period.
- **Dispute period and lockers as the sole defense against hot-key compromise.** All hot-quorum attacks (malicious withdrawals, hostile validator rotations, cold-set takeover) are recoverable only if lockers pause the bridge before the dispute period expires. Once the dispute period elapses, a finalizer can finalize malicious withdrawals (draining funds) or commit a validator set rotation that replaces cold keys, permanently eliminating the cold-quorum recovery path. There is no secondary on-chain defense: if lockers do not act within the dispute window, the damage is irrecoverable.
- **Locker threshold is set appropriately.** The `lockerThreshold` must be low enough that a sufficient number of honest lockers can trigger a pause during the dispute period. The threshold can only be changed by cold-quorum.
- **Hot/cold wallet separation.** Hot-key compromise is designed to be recoverable via cold-quorum `emergencyUnlock`, which atomically rotates the validator set, clears locker votes, and unpauses the bridge. Cold wallets can also invalidate malicious pending withdrawals and remove attacker-added lockers and finalizers. Cold-key compromise (>2/3 power) is the terminal failure mode with no on-chain recovery path.
- **L1 manages validator set correctness.** The contract does not validate validator addresses. The L1 is trusted to provide well-formed and correct validator sets.
- **L1 manages locker and finalizer lifecycle.** Validator set rotation does not automatically reconcile locker or finalizer roles. The L1 is responsible for issuing `modifyLocker` and `modifyFinalizer` calls to add new and remove retired lockers and finalizers.
- **Arbitrum sequencer honesty.** The dispute period relies on `block.timestamp` and `ArbSys.arbBlockNumber()`, both influenced by the Arbitrum sequencer. A compromised sequencer could shorten the effective dispute window or censor locker transactions.

Signed administrative messages (such as locker modifications, parameter changes, and withdrawal invalidations) carry no expiration timestamp or sequential nonce, meaning they remain valid indefinitely within a single epoch and can be submitted in any order.

Privileged Roles

- **Hot validator quorum (>2/3 power):** can request withdrawals, propose validator set updates, add lockers, and add finalizers.
- **Cold validator quorum (>2/3 power):** can emergency unlock (unpause and rotate validators), remove lockers, remove finalizers, change the dispute period, change the block duration, change the locker threshold, and invalidate withdrawals.
- **Lockers:** can vote to pause the bridge. The bridge pauses when the number of votes reaches the locker threshold.
- **Finalizers:** can finalize pending withdrawals and validator set updates after the dispute period has elapsed.

Medium Severity

M-01 `updateValidatorSet` Can Be Replayed to Block Finalization

The `updateValidatorSet` function constructs a deterministic message from the new epoch, validator addresses, and powers, but does not call `checkMessageNotUsed` and includes no nonce in the signed payload. Once a valid update transaction is broadcast, any third party can copy the calldata and re-submit it. Each call unconditionally overwrites `pendingValidatorSetUpdate`, refreshing `updateTime` and `updateBlockNumber`, which resets the dispute period checked by `finalizeValidatorSetUpdate`. This can be repeated indefinitely, preventing the validator set from being finalized through the `finalizeValidatorSetUpdate` and forcing the update through the `emergencyUnlock` route.

Consider adding replay protection by calling `checkMessageNotUsed(message)` inside the `updateValidatorSet` function or by otherwise ensuring that each signed update can only be processed once.

Update: *Resolved as recommended.*

Low Severity

L-01 Irrecoverable DoS Via Malicious Validator Set Parameters

Assuming that the `Bridge2` contract is not paused during the dispute period when malicious validator set updates are issued, a compromised hot quorum (>2/3 total power) can permanently brick the bridge, including the `emergencyUnlock` cold-wallet recovery path, through the following vectors, which could be mitigated by the on-chain code:

- **Epoch overflow**: `updateValidatorSet` places no upper bound on the epoch value. Setting it to `type(uint64).max` makes the `newValidatorSet.epoch >`

`activeValidatorSet.epoch` check unsatisfiable for all future updates, blocking validator rotations permanently.

- **Power overflow**: `checkNewValidatorPowers` only validates that cumulative power is greater than zero. Setting `totalValidatorPower` above `type(uint64).max / 3` causes the `3 * cumulativePower` computation in `checkValidatorSignatures` to overflow `uint64`, making the quorum check unreachable for any operation.
- **Validator count overflow**: no upper bound is enforced on the number of validators. An update with an excessively large validator array would make `checkValidatorSignatures` exceed the block gas limit, as the function iterates through the full set. The contract assumes 20-30 validators, but this is not enforced on-chain.

Consider enforcing a maximum epoch jump per update, so the epoch can never approach `uint64` max in practice, adding a cap in `checkNewValidatorPowers` ensuring total power does not exceed `type(uint64).max / 3`, and enforcing a maximum validator count consistent with the expected set size of 20-30.

Update: *Partially Resolved. The epoch and validator power overflow cases have been resolved. Validator count overflow has not been addressed as it is only reachable by a compromised quorum, and validator set well-formedness is an L1 trust assumption.*

L-02 Missing Lower Bound Validation in Locker Threshold Update

The `changeLockerThreshold` function does not enforce a lower bound on the new threshold value. Setting `lockerThreshold` to zero makes the pause condition `lockersVotingLock.length >= lockerThreshold` trivially true even when no lockers have voted, causing the bridge to pause immediately in the same transaction. Recovery requires calling `emergencyUnlock`, which forces a full validator set rotation and epoch bump, increasing the operational blast radius of what could be an accidental misconfiguration.

Consider requiring that `newLockerThreshold` is greater than 0 in the `changeLockerThreshold` function to prevent this scenario.

Update: *Resolved. The current code behaviour has been deliberately retained and documentation updated accordingly. The client stated:*

> *A zero threshold is the only cold-only pause entry: if all hot keys — hence all lockers — are lost, the cold quorum sets the threshold to 0 to pause the bridge, then runs `emergencyUnlock` to rotate in a new hot set. Requiring `> 0` would remove this last-*

> *resort recovery path and make total hot-key loss unrecoverable. The accidental self-pause it enables is itself recoverable via `emergencyUnlock`.*

L-03 Non-Standard EIP-712 Implementation

The `Bridge2` contract uses EIP-712 encoding for all signature-gated operations, including withdrawals, validator set updates, locker and finalizer management, and emergency unlock. However, the implementation deviates from the standard in several ways that collectively remove the signer's ability to inspect what they are authorizing.

All operations are signed using a single generic `Agent` struct with two fields: `source` (a string) and `connectionId` (a `bytes32`). The `makeMessage` function in `Bridge2` always sets `source` to the literal `"a"` and packs the actual action payload (pre-hashed via `keccak256`) into `connectionId`. This means a withdrawal request, a validator set rotation, and an emergency unlock all present identically to the signer's wallet UI: an `Agent` struct with a single-character string and an opaque 32-byte hash. The structured action parameters (such as recipient, amount, or new validator addresses) are not visible to the signer. Additionally, the domain separator uses `verifyingContract = address(0)` instead of the contract's own address, which removes the per-deployment binding that EIP-712 provides.

Consider redesigning the signing scheme so that signers can inspect the authorized action and target. Consider using explicit EIP-712 typed data per action, and consider setting the EIP-712 domain `verifyingContract` to the bridge address to improve compatibility with common signing UIs.

Update: *Resolved.*

L-04 Deposits Possible While Bridge Is Paused

The `Bridge2` contract declares a `Deposit` event, but the `depositWithPermit` function only emits `FailedPermitDeposit` on error paths and emits nothing on a successful USDC transfer. Deposit accounting instead relies on USDC `Transfer` events to credit users on L1. As a consequence, direct USDC transfers to the bridge address are also recognized as deposits even when the contract is paused, bypassing the intended halt of bridge activity.

Consider introducing a dedicated deposit function guarded by `whenNotPaused` that transfers assets from the user to the bridge and emits the `Deposit` event, ensuring both proper observability and pause enforcement.

Update: *Resolved by documenting the current design choice of accepting deposits even when the bridge is paused. Bidirectional pause solution (halting deposits as well as withdrawals when the contract is paused) remains a potential future update.*

L-05 Permit Front-Running Can Temporarily Block Deposits

The `depositWithPermit` function calls `usdcToken.permit` inside a try/catch block. If the permit call reverts, the function emits `FailedPermitDeposit` and returns without transferring any tokens. An attacker observing a pending `batchedDepositWithPermit` transaction can extract the permit signature and submit it directly to the USDC contract first. When the bridge's permit call executes, the user's permit nonce has already advanced, causing the call to revert and the deposit to silently fail.

The standard mitigation for permit front-running (falling back to `transferFrom` when the permit reverts) is not appropriate in this context. If a user holds a standing USDC allowance to the bridge from a previous `approve` call, the fallback `transferFrom` would succeed and pull funds for a deposit the user did not intend. Since `batchedDepositWithPermit` is operator-called, the impact is limited to a temporary operational inconvenience where the operator resubmits the failed deposits.

Consider documenting this as a known limitation so that operators and users are aware that affected deposits require new permit signatures.

Update: *Resolved by documenting the front-running attack risk and the reasoning behind not falling back to* `transferFrom` *if a permit call fails.*

Notes & Additional Information

N-01 Custom Cryptographic Helpers Lack Standard Validation Safeguards

The `Signature.sol` file implements custom free functions for EIP-712 domain separator construction (`makeDomainSeparator`), typed-data hashing (`hash`), and ECDSA signature

recovery (`recoverSigner`). The `recoverSigner` function forwards `(v, r, s)` directly to the `ecrecover` precompile and only checks that the recovered address is non-zero, without protecting against signature malleability. Similarly, `makeDomainSeparator` is called once in the constructor and cached as an immutable, so the domain separator becomes stale if the chain ID changes after a hard fork. Furthermore, the `\x19\x01` typed-data prefix is manually constructed rather than relying on a standard helper.

The project already integrates several OpenZeppelin contracts (`Pausable`, `ReentrancyGuard`, `SafeERC20`, `ERC20Permit`). Extending that pattern to cryptographic utilities would replace the custom implementations with battle-tested equivalents: `ECDSA.recover` protects from signature malleability, `EIP712` contract handles automatic domain separator recomputation on chain ID changes, and `MessageHashUtils.toTypedDataHash` standardizes the digest construction.

Consider replacing the custom cryptographic helpers in `Signature.sol` with OpenZeppelin's `ECDSA`, `EIP712`, and `MessageHashUtils` libraries to reduce implementation surface area and inherit well-audited validation logic.

Update: *Acknowledged, will resolve. The custom cryptographic helpers were retained. OpenZeppelin's ECDSA, EIP712, and MessageHashUtils libraries were not adopted, though this remains an optional future improvement.*

N-02 Naming Suggestions

Several identifiers in the `Bridge2` contract could be renamed to improve clarity:

- `checkMessageNotUsed` both validates that a message has not been previously consumed and marks it as used in the same call, but the name only reflects the validation half. A name like `consumeMessage` would more accurately describe the state mutation.
- `lockers` and `finalizers` are boolean mappings that would read more naturally as `isLocker` and `isFinalizer`, aligning with the convention already used by the `isVotingLock` function in the same contract.
- `lockersVotingLock` tracks addresses that have voted for an emergency pause, but the name is difficult to parse. A name like `emergencyLockVotes` would better convey its purpose.

Additionally, private and internal function names lack a leading underscore prefix, a widely adopted Solidity convention for visually distinguishing them from external and public functions.

Consider applying the renaming suggestions above and adopting a consistent naming convention for function visibility to improve the clarity and readability of the codebase.

Update: *Acknowledged, will resolve. The resolution has been deferred to a future non-behavioral cleanup, to avoid any changes in the ABI.*

N-03 Stale Data in `pendingValidatorSetUpdate` Variable After Finalization

The `finalizeValidatorSetUpdateInner` function copies all relevant fields from `pendingValidatorSetUpdate` into their respective storage variables (epoch, validator set hashes, total power, validator count) and then sets only `pendingValidatorSetUpdate.updateTime` to zero. The `PendingValidatorSetUpdate` struct contains seven fields, so after finalization six of them retain stale values from the previous update cycle.

Functionally, this does not cause issues because the `updateTime == 0` check guards against treating the struct as containing a valid pending update. However, the residual data occupies storage unnecessarily and could be a source of confusion.

Consider using `delete pendingValidatorSetUpdate` after emitting the event to fully clear all fields of the `pendingValidatorSetUpdate` variable.

Update: *Acknowledged, will resolve. A switch to `delete` is planned for a future cleanup.*

N-04 Missing Event Emissions And Indexed Event Parameters Reduce Off-Chain Observability

Throughout the codebase, several state-changing operations do not emit events:

- `voteEmergencyLock` adds the caller to the `lockersVotingLock` array without emitting an event for the individual vote.
- `unvoteEmergencyLock` removes the caller's vote through `removeLockerVote` without emitting any event. When `removeLockerVote` is called indirectly through `modifyLocker` during locker removal, the parent function emits `ModifiedLocker`, but that event signals the role change rather than the vote retraction.
- `addLockersAndFinalizers` sets both `lockers` and `finalizers` mappings for each initial validator without emitting `ModifiedLocker` or `ModifiedFinalizer` events. Since this function is only called in the constructor, the initial state can be

inferred from deployment arguments, but off-chain indexers relying on these events would miss the initial assignments.

As a result, off-chain monitoring of emergency lock vote state and initial role assignments requires polling rather than watching for events, which reduces observability of security-critical governance activity.

Additionally, several events declare no `indexed` parameters where they would aid off-chain filtering:

- `RequestedValidatorSetUpdate` and `FinalizedValidatorSetUpdate` lack an indexed `epoch`.
- `FailedWithdrawal` lacks an indexed `message`.

Consider emitting dedicated events for the operations listed above and adding `indexed` annotations to the mentioned event parameters to improve off-chain monitoring and filtering.

Update: *Acknowledged, will resolve. Fix has been deferred to a future coordinated observability update.*

N-05 Magic Numbers and String Literals Used in the Code

The `Bridge2` contract uses several inline numeric literals without named constants: `1337` and `31337` for local chain IDs, `100` for the ArbSys precompile address, and numeric error codes (`0`, `1`, `3`, `4`) in `getDisputePeriodErrorCode` and failure events.

Consider extracting these values into named constants or enums to improve readability and reduce the risk of inconsistencies, particularly for error codes.

Update: *Acknowledged, will resolve. The fix has been deferred to a future non-behavioral cleanup.*

Conclusion

`Bridge2` is a custodial USDC bridge connecting Arbitrum One to the TxFlow L1, secured by a BFT validator set with a layered hot/cold key architecture.

The audit identified issues primarily related to replay protection in validator set updates and input validation for critical parameters, both of which could impact bridge availability. The overall security model, with its role separation, dispute period, locker-based pause mechanism, and cold-quorum recovery path, provides a well-structured defense-in-depth approach against key compromise scenarios.

We would like to thank the TxFlow team for their active collaboration and responsiveness throughout the audit engagement.

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.