Code Assessment
of the Staking Final Smart Contracts

PUBLIC

Date June 16, 2026

Produced for

DUAL

By

CHAINSECURITY

Contents

1	Executive Summary	3
2	Assessment Overview	4
3	System Overview	5
4	Trust Model	10
5	System Considerations	12
6	Terminology	14
7	Findings	15
8	Limitations and use of report	20

1 Executive Summary

Dear Dual Foundation,

Thank you for trusting us to help you with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of Staking Final according to **Scope** to support you in forming an opinion on their security risks.

Dual Foundation operates Dual Network, an Arbitrum Orbit chain whose native coin is DUAL, on which holders stake native DUAL through the staking contract to receive the transferable xDUAL share token and a streamed share of protocol fees. The single contract in scope, `StakingFinal`, is an upgrade of the live staking proxy.

The most critical subjects covered in our audit are asset solvency, access control, and the state migration performed by the upgrade. Security regarding all these subjects is high.

The general subjects covered are the design of the reward streaming, transaction ordering, and upgradeability. Security regarding all these subjects is high.

In summary, we find that the codebase provides a high level of security. All issues raised in the initial review were addressed in **Version 2**, with the risk-accepted items all at informational severity.

It is important to note that security audits are time-boxed and cannot uncover all vulnerabilities. They complement but don't replace other vital measures to secure a project.

The following sections will give an overview of the system, our methodology, the issues uncovered, and how they have been addressed. We are happy to receive questions and feedback to improve our service.

Sincerely yours,
ChainSecurity

2 Assessment Overview

In this section, we briefly describe the overall structure and scope of the engagement, including the code commit which is referenced throughout this report.

2.1 Scope

The assessment was performed on the source code files inside the Staking Final repository based on the documentation files. The table below indicates the code versions relevant to this report and when they were received.

Version	Date	Commit Hash	Note
1	10 June 2026	977525ee359d060aadbaed1fbdf8d81f18398fa0	Initial review
2	14 June 2026	492bdda7c2583bd4b158e5e32bc772e78a69c753	Client fixes

For the Solidity smart contracts, the compiler version `0.8.35` was chosen.

```
src/
└── StakingFinal.sol
```

The in-scope `StakingFinal` is the upgrade implementation for the live UUPS staking proxy `0x69AD8a4eF0dDCD05A8b46053d04A5e20E7aDDcc1` on DUAL Network (chain id 6301). As read on-chain at block 825450, the proxy currently runs the legacy v1 implementation, is paused, has no stakers, both lifetime reward counters are zero, and it holds about 9.85 million DUAL parked in `pendingRewards`.

2.1.1 Excluded from scope

The remaining contracts in the repository are not part of this review. Tests, deployment scripts, off-chain tooling, and third-party dependencies under `lib/` are also out of scope. In particular, the OpenZeppelin Contracts Upgradeable library (v5.4.0) is assumed to be correct. The governance contract that consumes this voting power is not part of this repository or this review.

3 System Overview

This system overview describes the initially received version (**Version 1**) of the contracts as defined in the Assessment Overview. Each finding carries a version icon indicating the version it applies to.

Dual Foundation operates DUAL Network, an Arbitrum Orbit chain whose native coin is DUAL. Protocol fees collected across the system are routed through a `FeeDispatcher` to a staking contract, where holders of native DUAL stake to receive a streamed share of those fees. The single contract in scope is `StakingFinal.sol`, the upgrade implementation for the currently deployed staking UUPS proxy.

3.1 Contract Structure and Deployment

The `StakingFinal` contract is an ERC-20 token (name `Staked DUAL`, symbol `xDUAL`) extended through OpenZeppelin v5 upgradeable modules:

Module	Role in the contract
`ERC20Upgradeable`	The xDUAL share token, minted 1:1 against staked DUAL.
`ERC20PermitUpgradeable`	EIP-2612 gasless approvals. The EIP-712 domain is initialized during migration.
`ERC20VotesUpgradeable`	Vote checkpoints over xDUAL balances, using a timestamp clock.
`Ownable2StepUpgradeable`	Two-step ownership. The owner upgrades, pauses, and configures.
`PausableUpgradeable`	Pausing blocks new stakes only.
`ReentrancyGuardUpgradeable`	Guards every state-changing entry point that moves native value.
`UUPSUpgradeable`	Owner-gated implementation replacement.

`StakingFinal` uses default Solidity flat storage sequential slots 0 to 12. The OpenZeppelin modules use ERC-7201 namespaced storage and do not collide with these slots. A trailing `uint256[42]` gap reserves space for future state. The full slot mapping (frozen from the deployed proxy) is documented under Storage Layout and Migration.

The owner is a privileged role with full control over upgrades and configuration. The `feeDispatcher` is the only address allowed to push fee rewards into the contract through `receive()`. A complete role analysis is in the Trust Model.

3.2 Staking and the xDUAL Share Token

Any native DUAL holder can stake through the payable `stake()` function, which mints xDUAL 1:1 against `msg.value`. An xDUAL holder can unstake using `unstake(amount)`, which burns xDUAL and returns native DUAL 1:1 via a `call()` native value transfer.

The contract also maintains a `totalStaked` counter that tracks principal. This counter mirrors `totalSupply()` and is not read by any reward, claim, or streaming calculation. It exists only to keep the storage layout consistent with the deployed proxy.

xDUAL is a fully transferable ERC-20 token. Transfers move both reward accrual and voting power between the parties, as described in Reward Accrual and Voting Power and Delegation.

3.2.1 Staking lifecycle

Function	Effect
`stake()`	Mints xDUAL 1:1 with `msg.value`. Folds any `pendingRewards` into a new stream. Blocked while paused.
`unstake(amount)`	Burns xDUAL, returns principal 1:1. Accrues rewards but does not pay them. Open while paused.
`claimRewards()`	Pays accrued native DUAL rewards. Open while paused.
`exit()`	Burns the caller's full balance (if any) and pays accrued rewards in one transaction, reverting only if both are zero. Open while paused.

All four functions are `nonReentrant`. The three that send native DUAL (`unstake()`, `claimRewards()`, `exit()`) update state before the external `call` and revert the whole transaction if the transfer fails.

3.3 Reward Accrual

Rewards are native DUAL streamed to stakers in proportion to their xDUAL balance over time, using the Synthetix staking-rewards accumulator pattern.

The global accumulator `rewardPerTokenStored` records cumulative reward per xDUAL, scaled by `PRECISION` (10^{18}). The live value including the unsnapshot portion of the active stream is

$$\text{rewardPerToken} = \text{rewardPerTokenStored} + \frac{(\text{lastTimeRewardApplicable} - \text{lastUpdateTime}) \cdot \text{rewardRate} \cdot \text{PRECISION}}{\text{totalSupply}}$$

where `lastTimeRewardApplicable` is `min(block.timestamp, periodFinish)`. When `totalSupply()` is zero the accumulator does not advance.

For each account, `userRewardPerTokenPaid` stores the `rewardPerToken` value at the account's last update and `userAccruedRewards` stores the unclaimed reward amount. The internal `_updateReward(account)` first advances the global accumulator, then credits the account with

$$\text{shares} \cdot \frac{\text{rewardPerTokenStored} - \text{userRewardPerTokenPaid}}{\text{PRECISION}}$$

and updates the account snapshot.

`_updateReward()` is invoked from the ERC-20 `_update()` hook for the sender and the receiver on every mint, burn, and transfer and `exit()`. `previewRewards(user)` is a view function that computes the same value.

3.4 Reward Streaming and Accounting

Fee rewards are injected as native DUAL through two paths:

1. the `feeDispatcher` sends fees through `receive()`
2. the owner injects bonuses through `addBonus()`.

Both route into `_ingestRewards()` that:

- If `totalSupply()` is zero or the contract is paused, the funds are added to `pendingRewards` and held.
- Otherwise the funds plus any existing `pendingRewards` are passed to `_notifyReward(amount)`, which (re)starts the stream.

`_notifyReward(amount)` computes a new `rewardRate` from the current period's un-streamed amount (`leftover`) plus the new amount, and starts a fresh period ending at `block.timestamp + rewardsDuration`. The remainder that does not divide evenly (`dust`) is parked back into `pendingRewards` for the next notification. Each notification resets the reward period, including the notification issued by `stake()`, which re-schedules the active stream.

3.5 Pausing

`pause()` and `unpause()` are owner functions. Pausing blocks the `stake()` function only. Fees and bonuses that arrive while paused are parked into `pendingRewards` and not streamed. An already-active stream continues to run during the pause. `unpause()` schedules any parked rewards if supply is non-zero.

3.6 Voting Power and Delegation

xDUAL is a governance token through `ERC20VotesUpgradeable`. Voting power is checkpointed and queried over time. The contract reports a timestamp clock (`clock()` returns `block.timestamp`, `CLOCK_MODE()` returns `mode=timestamp`).

Under `ERC20Votes`, an account's balance counts as voting power only after the account delegates. `StakingFinal` auto-delegates any holder with unset delegatee to itself by default on first token receive.

Every xDUAL holder has voting power by default. The zero-address delegate has two meanings that the `_update()` hook cannot distinguish: an account that has never delegated, and an account that deliberately set its delegate to the zero address to hold no voting power. The auto-delegation re-applies self-delegation to both.

3.7 Storage Layout and Migration

`StakingFinal` is installed behind a proxy that was already initialized at version 1. Its storage layout is fixed to match the deployed implementation. Slots 0 to 11 keep their deployed types and positions. The new field `committedRewards` is appended at slot 12, which was previously reserved gap and reads zero on the live proxy. The reserved gap shrinks from `uint256[43]` to `uint256[42]`.

Two initialization paths exist. `initialize()` runs once for a hypothetical fresh deployment and is gated by the `initializer` modifier, so it cannot run on the already-initialized proxy. `reinitializePermit()` is the migration entry point for the live proxy. It is `onlyOwner` and gated by `reinitializer(2)`, so it runs at most once. It performs two actions:

1. It initializes the `ERC20Permit` (EIP-712) domain to (`Staked DUAL`, `1`). The deployed version 1 never inherited `ERC20Permit`, so this domain was unset. The same domain is shared by `ERC20Votes` signature-based delegation.
2. It seeds `committedRewards` to `lifetimeRewardsScheduled - lifetimeRewardsClaimed`, the live outstanding owed rewards computed from the preserved version 1 counters.

3.8 Assumptions

- The contract is deployed on an EVM chain (DUAL Network, an Arbitrum Orbit chain) whose native coin is the DUAL asset being staked. All value movements use native `call` transfers.
- The `feeDispatcher` address is trusted to forward legitimate fee amounts. Only that address may call `receive()`.

- The owner is trusted. The owner can replace the implementation with arbitrary code through `upgradeToAndCall()`, which is the only remaining path that can move staker funds after the removal of the version 1 emergency withdrawal.
- Stakers that are contracts must be able to receive native DUAL, or hold transferable xDUAL that can be moved to an account that can. A staker whose `receive()` reverts cannot be paid by `unstake()`, `claimRewards()`, or `exit()`.
- The governance system that reads xDUAL voting power is external to this scope and is assumed to handle the delegation semantics described above.
- DUAL force-sent to the contract (for example through a self-destruct) is treated as reward surplus and cannot be recovered by any party other than stakers.

3.9 Function Summary

3.9.1 State-Modifying Functions

Function	Description
`stake()`	Stake native DUAL, mint xDUAL 1:1, fold pending rewards into a stream. Blocked while paused.
`unstake(amount)`	Burn xDUAL, return principal 1:1.
`claimRewards()`	Pay accrued native DUAL rewards to the caller.
`exit()`	Burn the full xDUAL balance and pay accrued rewards in one call.
`addBonus()`	Owner injects a discretionary native DUAL bonus into the reward flow.
`setFeeDispatcher(addr)`	Owner sets the authorized fee sender. Rejects zero and no-op.
`setRewardsDuration(d)`	Owner sets the streaming period. Only when no period is active.
`pause()` / `unpause()`	Owner pauses new stakes / resumes and schedules parked rewards.
`initialize(...)`	One-time setup for a fresh deployment. Not reachable on the live proxy.
`reinitializePermit()`	One-time migration. Initializes the permit domain and seeds `committedRewards`.
`permit(...)`	EIP-2612 gasless approval of xDUAL.
`delegate(...)` / `delegateBySig(...)`	Standard `ERC20Votes` delegation.
`transfer()`, `transferFrom()`, `approve()`	Standard ERC-20 operations on xDUAL.
`receive()`	Accepts fee DUAL from the `feeDispatcher` only.
`upgradeToAndCall(impl, data)`	Owner replaces the implementation.

3.9.2 View Functions

Function	Description
`rewardPerToken()`	Live global reward-per-token accumulator.
`previewRewards(user)`	Claimable rewards for `user` right now.
`lastTimeRewardApplicable()`	`min(block.timestamp, periodFinish)`.
`committedRewards()`	Outstanding owed rewards.
`lifetimeRewardsScheduled()` / `lifetimeRewardsClaimed()`	Analytics counters.
`pendingRewards()`, `rewardRate()`, `periodFinish()`, `rewardsDuration()`, `lastUpdateTime()`	Stream state.
`totalStaked()`, `feeDispatcher()`	Principal mirror and configured fee sender.
`getVotes()`, `getPastVotes()`, `delegates()`, `clock()`, `CLOCK_MODE()`, `nonces()`, `DOMAIN_SEPARATOR()`	Governance and permit views.
`balanceOf()`, `totalSupply()`, `allowance()`, `name()`, `symbol()`, `decimals()`	Standard ERC-20 views.

3.10 Changes in Version 2

Two code changes were applied between **Version 1** and **Version 2**, addressing findings raised in the initial review.

3.10.1 Inflow counter fix and rename (storage slot 4)

The analytics counter in the 4th storage slot was previously incremented inside `_replaceLeftoverRewards` by `scheduled - leftover` on each notification, so rewards that were parked and later re-scheduled were counted again. The increment is now performed once in `_ingestRewards`, at the single external-inflow choke point, counting the full received `amount` exactly once at receipt (covering both `receive()` fees and `addBonus()`). The field and getter were renamed from `lifetimeRewardsScheduled` to `lifetimeRewardsReceived` to match the corrected semantics. The `reinitializePermit()` adds the parked `pendingRewards` to `lifetimeRewardsReceived`, so the counter starts at the true total ever received on the live proxy. See lifetimeRewardsScheduled over-States Outstanding Rewards After a Shrink or Empty-Pool Park.

3.10.2 `stake()` no longer re-schedules an active stream

The `stake()` fold of `pendingRewards` into `_notifyReward` is now gated on `block.timestamp >= periodFinish`, so a `stake()` can only bootstrap a stream when none is active. Parked dust accrued during an active period is folded by the next legitimate fee inflow (`receive()` or `addBonus()`) or once the current period ends. See Re-Scheduling a Reward Stream Can Lower the Effective Rate and Extend the Period.

4 Trust Model

4.1 Upgrade Assumptions

The review of `StakingFinal` assumes the following deployment procedure, which is what the in-repo `script/UpgradeStaking.s.sol` performs:

- The live proxy currently runs the legacy `StakingV1` implementation (`src/legacy/StakingV1.sol`) - storage slots 0 to 11 hold the v1 layout (`rewardPerTokenStored`, `userRewardPerTokenPaid`, `userAccruedRewards`, `totalStaked`, `totalFeesDispatched`, `totalRewardsClaimed`, `feeDispatcher`, `rewardRate`, `lastUpdateTime`, `periodFinish`, `rewardsDuration`, `pendingRewards`), and slot 12 is in the reserved gap. This was confirmed on-chain at block 825450 (chain id 6301), where `committedRewards()` reverts on the proxy and the v1 getters resolve.
- The owner upgrades the proxy directly from `StakingV1` to `StakingFinal`. The owner does not upgrade to or via the new canonical `src/Staking.sol` implementation, which has an incompatible slot 3.
- The upgrade is executed atomically as `proxy.upgradeToAndCall(newImpl, abi.encodeCall(StakingFinal.reinitializePermit, ()))`. The implementation pointer swap and `reinitializePermit()` run in the same transaction, so the proxy is never live with an uninitialised ERC20Permit domain or an unseeded `committedRewards`. A bare `upgradeTo` call that skips `reinitializePermit()` is never performed.
- The live v1 state satisfies `totalFeesDispatched >= totalRewardsClaimed` at the moment of upgrade, so the `reinitializePermit()` seed `committedRewards = lifetimeRewardsScheduled - lifetimeRewardsClaimed` does not underflow.

4.2 Roles

4.2.1 Owner

- Trust level: fully trusted.
- Holds upgrade rights (`_authorizeUpgrade`), can pause/unpause, set the `FeeDispatcher`, change `rewardsDuration` (only between streams), and inject bonus rewards via `addBonus()`.
- Two-step ownership transfer via `Ownable2StepUpgradeable`.
- Worst case: upgrades the implementation to malicious code and drains the contract, or pauses indefinitely. Pausing does not block `unstake()` or `claimRewards()`, so stakers can always exit even while paused.

4.2.2 FeeDispatcher

- Trust level: partially trusted.
- The only account allowed to send native DUAL to `receive()` as fee rewards.
- Worst case: withholds or delays fee deliveries. Cannot mint xDUAL or claim rewards on behalf of stakers. The address is settable by the owner.

4.2.3 Stakers (end users)

- Trust level: untrusted.
- Can `stake()`, `unstake()`, `claimRewards()`, `exit()`, transfer xDUAL, and use ERC20Permit / ERC20Votes delegation.
- xDUAL is minted 1:1 with DUAL and is freely transferable, so reward accrual follows the recipient on transfer.

4.3 External Dependencies

- Native DUAL is the only asset held - no ERC-20 transfers, no external price oracles, no bridges read by this contract.
- L2 / Orbit chain assumption: `clock()` returns `block.timestamp` (EIP-6372 `mode=timestamp`) because block numbers on Arbitrum/Orbit are unreliable for governance checkpointing. The deployment chain is assumed to provide monotonically increasing timestamps consistent with EVM semantics.
- OpenZeppelin upgradeable v5 contracts (`ERC20Upgradeable`, `ERC20PermitUpgradeable`, `ERC20VotesUpgradeable`, `Ownable2StepUpgradeable`, `PausableUpgradeable`, `ReentrancyGuardUpgradeable`, `UUPSUpgradeable`) are assumed to be correct and used at compatible versions.

5 System Considerations

We leverage this section to highlight findings that are not necessarily issues. The mentioned topics serve to clarify or support the report, but do not require a modification inside the project. Instead, they should raise awareness in order to improve the overall understanding for users and developers.

SC1 First Staker Captures Full Parked Reward Stream After the Upgrade

System Consideration | Version 1

When the contract holds parked `pendingRewards` while `totalSupply()` is zero, the first account to stake starts the reward stream and, for as long as it remains the only staker, accrues the entire stream. An on-chain read of the live proxy shows that `totalSupply` is zero and `pendingRewards` holds about 9.85 million DUAL. The contract is currently paused, so the capture triggers only once the owner unpauses and the first stake occurs. The first staker after the upgrade will earn 100% of the reward stream until more stakers arrive. The advantage is driven by reaction time rather than staked capital, because even a 1 wei stake accrues 100 percent of the stream while the account is alone.

Acknowledged

Client response:

Addressed by process, not code. The about 9.85M DUAL in `pendingRewards` streams linearly over the 7-day `rewardsDuration` once the first stake folds it into a stream. A staker that is alone therefore captures only `rewardRate * (time alone)`, a fraction `(time alone) / 7 days` of the pool, independent of stake size. The exposure is bounded by the alone-window, not by capital. For example, 10 minutes alone caps capture at about `0.10%` of the pool, 1 hour at about `0.60%`, and 1 day at about `14%`.

The mitigation is to keep the alone-window to minutes. The protocol will publish a public countdown to a synchronized opening time and `unpause()` at that moment, so a broad set of users stake within the first block(s). Stakers landing in the same block share the stream strictly pro-rata by capital (reward accrual is time-based, so zero time elapses between same-block stakes and no first-mover advantage accrues within a block). As insurance against low turnout, the treasury may additionally seed a stake in the same transaction as `unpause()`, which caps any external first-mover's capture regardless of transaction ordering. At a 7-day duration, a realistic few-minute window bounds the leak to well under `0.1%` of the pool.

SC2 Stakers Must Be Able to Receive Native DUAL

System Consideration | Version 1

`unstake()`, `claimRewards()`, and `exit()` in `StakingFinal.sol` pay out with a native value transfer through `call` to `msg.sender`, and revert the whole transaction when the transfer fails. A staker whose `receive()` or `fallback()` rejects native DUAL cannot withdraw through any of these functions.

Acknowledged

By design. Payout via `call` with revert-on-failure is preferred to silently stranding funds. xDUAL is freely transferable, so a contract that cannot receive native DUAL can move its shares to one that can. This behaviour is documented for integrators.

> Note that transferring shares does not transfer already accrued rewards. Thus, if a contract is not able to claim native tokens, it should move its xDUAL balance to another contract immediately, so as not to accrue irretrievable rewards.

6 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- **Likelihood** represents the likelihood of a finding to be triggered or exploited in practice
- **Impact** specifies the technical and business-related consequences of a finding
- **Severity** is derived based on the likelihood and the impact

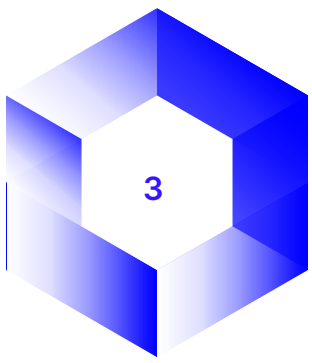
We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

Likelihood	**Impact**		
	High	Medium	Low
High	**Critical**	**High**	**Medium**
Medium	**High**	**Medium**	**Low**
Low	**Medium**	**Low**	**Low**

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

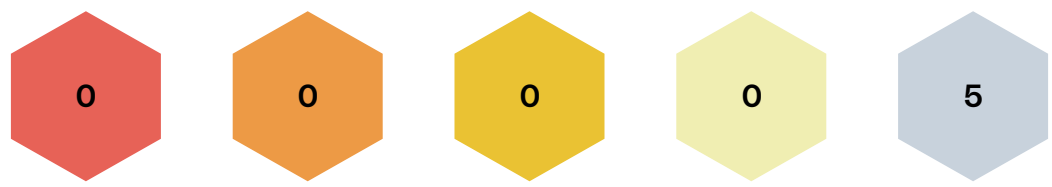
7 Findings

In this section, we describe the findings identified during the course of the engagement. The findings are categorized by severity as explained in the Terminology section. Below we provide an overview of the total number of findings per severity, as well as the number of open issues.



Open Issues

Total number of findings



Critical | High | Medium | Low | Informational

7.1 Overview

The following table lists all identified findings along with their severity and current status. A finding is considered resolved once it has been fully corrected or the specification has been changed accordingly. Findings with any other status, including partial fixes, acknowledgements, and accepted risks, remain open.

Info	#001	Reward Duration Cannot Be Updated While a Stream Is Active	Risk Accepted
Info	#002	A Small Reward Notification Closes an Active Reward Period	Risk Accepted
Info	#003	lifetimeRewardsScheduled over-States Outstanding Rewards After a Shrink or Empty-Pool Park	Code Corrected
Info	#004	Migration Changes the EIP-712 Domain Used for Vote Delegation Signatures	Risk Accepted
Info	#005	Re-Scheduling a Reward Stream Can Lower the Effective Rate and Extend the Period	Code Corrected

7.2 Descriptions

#001 Reward Duration Cannot Be Updated While a Stream Is Active

Informational	Version 1	Risk Accepted

`setRewardsDuration` reverts when `block.timestamp < periodFinish`. Every fee inflow through `receive()` or `addBonus()` calls `_notifyReward`, which resets `periodFinish` to `block.timestamp + rewardsDuration`. While fees arrive at intervals shorter than `rewardsDuration`, `periodFinish` is continually pushed forward, so the owner cannot change `rewardsDuration`.

The reset is not limited to organic fee flow. As described in Re-Scheduling a Reward Stream Can Lower the Effective Rate and Extend the Period, any `stake()` that folds parked `pendingRewards` also calls `_notifyReward` and pushes `periodFinish` out, so the lock can be sustained permissionlessly, not only under heavy fee traffic.

Because `stake()` can re-arm the lock, suspending fee inflows alone is insufficient. The owner must pause the contract, which blocks `stake()` and parks inflows, then wait for the active period to finish before `setRewardsDuration` becomes callable.

In **Version 2**, the permissionless period extension was removed due to Re-Scheduling a Reward Stream Can Lower the Effective Rate and Extend the Period. Only the owner-facing operational constraint described above remains (pause, wait out the active period, then call `setRewardsDuration`), which is why this issue is downgraded from **Version 1** low to **Version 2** informational.

> **Risk Accepted**
>
> The owner-facing constraint is intended. To change the duration, the owner pauses (which parks inflows and blocks new stakes) and waits out the active period, then calls `setRewardsDuration`. The permissionless lock re-arming vector (any `stake()` folding parked rewards) was removed in commit `4e7cf3b`, see Re-Scheduling a Reward Stream Can Lower the Effective Rate and Extend the Period, so the lock can no longer be indefinitely sustained without owner action.

#002 A Small Reward Notification Closes an Active Reward Period

Informational	Version 1	Risk Accepted

In `_notifyReward`, when `amount + leftover` is smaller than `rewardsDuration` in wei, the rate division floors to zero.

```
newRate = totalToSchedule / rewardsDuration; // floor, zero when totalToSchedule <
    rewardsDuration
scheduled = newRate * rewardsDuration;       // zero
dust = totalToSchedule - scheduled;          // the whole amount
```

With `newRate` zero, `scheduled` is zero. The whole `totalToSchedule` is parked into `pendingRewards` and the reward period is closed early. Only the `RewardsParked` event is emitted in this case, not `RewardsNotified`.

Reachability

While a period is active at least one second remains, so `leftover` is at least `rewardRate`. The closure condition `amount + leftover < rewardsDuration` therefore requires `rewardRate < rewardsDuration`. For a 7 day duration (`604800` seconds) that bounds the period's total scheduled reward below `604800 * 604800 = 365,783,040,000` wei, about `3.66e-7` DUAL. Any larger stream has a per-second `leftover` far above

`rewardsDuration`, so `newRate` stays at least one and the notification re-amortises the remainder instead of closing it.

Impact

The early closure is reachable only for dust-scale pools whose entire scheduled reward is below `rewardsDuration` squared in wei, about `3.66e-7` DUAL for a 7 day duration, so it has no practical economic effect. No funds are at risk. `_updateReward(address(0))` credits rewards streamed up to the closing block, and `committedRewards` and `pendingRewards` are conserved, with `committedRewards >= leftover` always holding so the decrement cannot underflow. The terminated remainder is deferred to the next notification. The observable effect is an active period that ends without a `RewardsNotified` event. The general re-scheduling behaviour for non-dust streams is described in Re-Scheduling a Reward Stream Can Lower the Effective Rate and Extend the Period.

Risk Accepted

Confirmed. The early-closure branch is only reachable for dust-scale pools whose entire scheduled reward is below `rewardsDuration` squared in wei, about `3.66e-7` DUAL for a 7 day duration, so it has no practical economic effect. No funds are at risk. `committedRewards` and `pendingRewards` are conserved, and the terminated remainder is deferred to the next notification. The observable effect (an active period ending without a `RewardsNotified` event) is accepted.

#003 lifetimeRewardsScheduled over-States Outstanding Rewards After a Shrink or Empty-Pool Park

Informational | **Version 1** | **Code Corrected**

`reinitializePermit` seeds `committedRewards` to `lifetimeRewardsScheduled - lifetimeRewardsClaimed`, but `lifetimeRewardsScheduled` is increment-only while `committedRewards` is also decremented whenever un-streamed rewards are parked back, through `_parkRemainingRewardsIfEmpty` when the last staker leaves or through the shrink and early-close branches of `_replaceLeftoverRewards`. Parked DUAL that is later re-scheduled increments `lifetimeRewardsScheduled` a second time, so the same rewards are double counted and the counter can exceed the total fees the contract has ever received. After the first park the seed identity no longer holds, and `lifetimeRewardsScheduled - lifetimeRewardsClaimed` over-states the live outstanding amount. The reward logic reads only `committedRewards`, which stays correct, so the effect is limited to analytics and to any off-chain consumer or future re-seed that trusts the two counters.

Code Corrected

The counter increment was moved out of the streaming path and into the single external-inflow choke point `_ingestRewards`, where the full received `amount` is counted exactly once at receipt (covering both `receive()` fees and `addBonus()`), regardless of any later park or re-schedule. The slot-4 field and getter were renamed `lifetimeRewardsScheduled` to `lifetimeRewardsReceived` to match the corrected semantics. No reward, claim, or solvency logic was touched. `committedRewards` remains the sole field read by `_availableForRewards`, and its updates are byte-for-byte unchanged.

The migration in commit `492bdda` additionally re-bases `lifetimeRewardsReceived` from v1's net figure to gross by adding the parked `pendingRewards` (on the live proxy, the about 9.85M park), so the counter starts at the true total ever received.

#004 Migration Changes the EIP-712 Domain Used for Vote Delegation Signatures

Informational | **Version 1** | **Risk Accepted**

`reinitializePermit` initializes the `ERC20Permit` EIP-712 domain to (`Staked DUAL`, `1`). That domain is shared by `ERC20Votes` for `delegateBySig`. The deployed version 1 never inherited `ERC20Permit` and never initialized the domain, so version 1 computed `delegateBySig` digests against an uninitialized domain. After the migration the domain is populated, which changes the digest that both `delegateBySig` and `permit` expect. Thus, any `delegateBySig` signature created against the version 1 proxy is invalid after the upgrade, and any signature produced afterward must use the (`Staked DUAL`, `1`) domain.

Risk Accepted

Initializing the EIP-712 domain to (`Staked DUAL`, `1`) is the intended purpose of the migration. The live proxy currently has no stakers and no outstanding xDUAL, so no live delegation signatures exist that would be invalidated. Post-migration signing follows the standard (`Staked DUAL`, `1`) domain.

#005 Re-Scheduling a Reward Stream Can Lower the Effective Rate and Extend the Period

Informational | **Version 1** | **Code Corrected**

`_notifyReward()` runs on every fee inflow through `receive()` and `addBonus()`, and on every `stake()` that folds pending rewards. Because each notification parks a `dust` remainder back into `pendingRewards`, even a 1 wei stake might re-trigger it permissionlessly.

Rate is not monotonic in the amount notified

Because `periodFinish` is reset a full `rewardsDuration` ahead on each call, the un-streamed `leftover` is re-amortised over a fresh duration every time. The new rate can be lower than the rate it replaces, even when the notification adds rewards.

Consider a stream of 100 DUAL over a 1 week duration. The rate is 100 DUAL per week. After half a week, 50 DUAL has streamed and 50 DUAL is the un-streamed `leftover`, on track to finish over the remaining half week. A `stake()` then occurs:

- `leftover` is 50, so `totalToSchedule` is 50.
- `rewardRate` becomes about 50 DUAL per week.
- `periodFinish` is reset to `block.timestamp + 1 week`.

Thus, `stake()` drops the near-term rate from 100 to about 50 DUAL per week and moves the end of the distribution from half a week away to a full week away.

The effect is largest near expiry. The closer a notification lands to `periodFinish`, the larger `leftover` is relative to a fresh full duration. This is also why `setRewardsDuration` is effectively never callable while inflows continue, see Reward Duration Cannot Be Updated While a Stream Is Active. When `amount + leftover` is too small to yield a non-zero rate, the period is closed instead of re-scheduled, see A Small Reward Notification Closes an Active Reward Period.

A staker can repeat dust stakes while `pendingRewards` is non-zero to keep pushing `periodFinish` forward and slow the stream for everyone. The action is cheap on an L2, gives the actor no profit, and only delays distribution.

Impact

No principal or reward funds are at risk: `_updateReward(address(0))` credits already-streamed rewards to current stakers before each rate change, and `committedRewards` and `pendingRewards` are conserved. The harm is a reward rate that is not monotonic in the amount added and a distribution that any account can stretch out indefinitely, for free, by re-triggering `_notifyReward` through `stake()`.

Code Corrected

The `stake()` fold of `pendingRewards` is now gated on `block.timestamp >= periodFinish`:

```
if (pendingRewards > 0 && block.timestamp >= periodFinish) {
    uint256 rewardsToNotify = pendingRewards;
    pendingRewards = 0;
    _notifyReward(rewardsToNotify);
}
```

A `stake()` can therefore only bootstrap a stream when none is active and can no longer re-notify a live stream. Parked dust accrued during an active period is folded by the next legitimate fee inflow (`receive()` or `addBonus()`) or once the current period ends. The bootstrap path (first stake into an idle or parked pool, for example immediately after the upgrade) is unchanged. Organic rescheduling on genuine fee inflows is intentionally retained.

8 Limitations and use of report

Security assessments cannot uncover all existing vulnerabilities; even an assessment in which no vulnerabilities are found is not a guarantee of a secure system. However, code assessments enable the discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary. In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it. Some of the issues may affect the entire application, while some lack protection only in certain areas. This is why we carry out a source code assessment aimed at determining all locations that need to be fixed. Within the customer-determined time frame, ChainSecurity has performed an assessment in order to discover as many vulnerabilities as possible.

The focus of our assessment was limited to the code parts defined in the engagement letter. We assessed whether the project follows the provided specifications. These assessments are based on the provided threat model and trust assumptions. We draw attention to the fact that due to inherent limitations in any software development process and software product, an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third-party infrastructure which adds further inherent risks as we rely on the correct execution of the included third-party technology stack itself. Report readers should also take into account that over the life cycle of any software, changes to the product itself or to the environment in which it is operated can have an impact leading to operational behaviors other than those initially determined in the business specification.