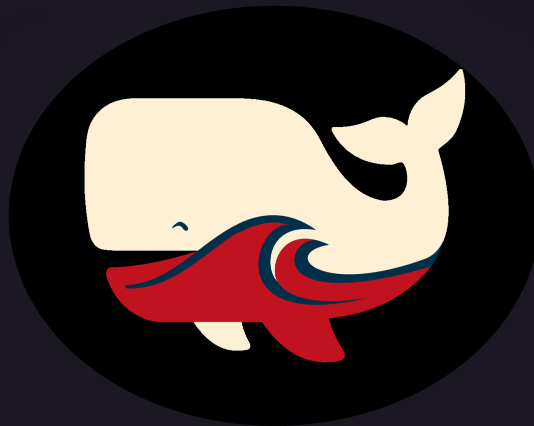
SHERLOCK

Security Review For Boardwalk



Collaborative Audit Prepared For: **Boardwalk**
Lead Security Expert(s): **LZ_security**
Date Audited: **April 25 - May 6, 2026**

Introduction

Boardwalk is a permissionless, community-centered fee-protection protocol for launching, discovering, and participating in token economies. Issuers create a launch, define its rules up front, and auction a new token under visible mechanics. If the auction reaches its graduation threshold, the raised asset pairs with the token supply to seed permanently locked liquidity, and the token moves into a live economy. If the threshold isn't met, contributors can claim a full refund.

The protocol facilitates launches end-to-end in one place: auction, liquidity seeding, LP participation, fee routing, and vote direction. Issuers choose between two launch paths – Express, a 24-hour auction with streamlined defaults, and Advanced, a 7-day auction with configurable vesting, multiple fee recipients, and an optional referrer. Liquidity providers of tokens launched through Boardwalk can earn from up to three sources: normal swap fees, a dedicated fee-stream from all token swaps and transfers, and a share of the vesting allocation if vesting was configured into the launch. Liquidity providers also earn Participation Points the longer they stay staked, which increase their relative claim on fee and vesting streams over time.

Economies launched through Boardwalk include fee protections at the token level. The fee is built into the token itself, so it applies to onchain movement across any venue where the token trades – not just a single app. This makes it uneconomical for outside actors using sophisticated strategies to take fees away from the intended economy. Once the launch is live, core fee percentages are fixed by protocol design.

Boardwalk is also where the people who support a launch over time come together. Café Boardwalk, a public web forum tied to every launch on the protocol, gives contributors, issuers, and third-party supporters – liquidity providers, growth contributors, security reviewers, treasury-management services, and public-good contributors – a built-in space to ask questions, post updates, and coordinate. The Café is a discussion surface, not an onchain mechanism: participation there does not change launch rules, fee routing, vesting, or voting weight.

BMX is Boardwalk's deflationary protocol token. BMX functions as a consumption token, burned by users across all chains to launch a token economy and influence project discovery through Upvoting and Downvoting. BMX can be staked to vote-direct a portion of protocol fees. Each weekly epoch, stakers vote-direct designated fees toward one of four outcomes – BMX Buy & Burn, Route to Operations Reserve, Perma-Lock BMX/WETH liquidity, and Route to BMX Stakers. Voting is winner-take-all, with votes in one epoch directing the fees collected in the next.

Scope

Repository: morphex-labs/boardwalk-contracts

Audited Commit: ecc5757a4522a7fa504ec86a06b0b337eda50f9f

Final Commit: 0b81f1792d3d39ec97b897a4316180cf231a5327

Files:

- src/base/AllocationLib.sol
- src/base/MembershipDiscount.sol
- src/base/Timelocked.sol
- src/core/BoardwalkFeeCollector.sol
- src/core/BoardwalkLPManager.sol
- src/core/BoardwalkToken.sol
- src/core/FeeDistributor.sol
- src/core/LaunchFactory.sol
- src/core/LPStaking.sol
- src/core/PresaleManager.sol
- src/core/VestingStream.sol
- src/governance/GovernanceVoter.sol
- src/governance/LPLocker.sol
- src/governance/ParticipationDistributor.sol

Final Commit Hash

0b81f1792d3d39ec97b897a4316180cf231a5327

Findings

Each issue has an assigned severity:

- High issues are directly exploitable security vulnerabilities that need to be fixed.
- Medium issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- Low/Info issues are non-exploitable, informational findings that do not pose a security risk or impact the system's integrity. These issues are typically cosmetic or related to compliance requirements, and are not considered a priority for remediation.

Issues Found

High	Medium	Low/Info
0	5	14

Issues Not Fixed and Not Acknowledged

High	Medium	Low/Info
0	0	0

Audit Methodology used for Boardwalk

1. Phased Approach

The Sherlock team's security review proceeded through four phases.

1.1 Orientation

The first phase was a complete read of the project specification and a structural scan of all files. The goal was a working mental model and diagram of the protocol: the lifecycle of a launch, the flow of fees through the system, the privilege hierarchy, and the governance loop before delving into our line-by-line review. Notes from this phase captured the per-launch versus singleton split, the canonical interaction sequence (presale → seed → staking + vesting → fee distribution → governance), and the points at which clones inherit state versus configuration.

1.2 Contract Mapping

The second phase mapped each contract individually. For every contract the review identified:

- All external entry points, separated into permissionless and privileged.
- The privilege ladder: initializer, owner, keeper, factory, fee collector, fee distributor, presale manager, issuer, recipient, and so on, including which actions each role can take.
- All cross-contract callbacks and their atomicity guarantees, with particular attention to revert-handling boundaries (`try/catch` versus propagating).
- Storage layout for upgradeability and clone-safety considerations, including initializer locks and slot conflicts.

This phase produced the structured entry-point catalogue and the cross-contract call graph that the deep-dive phase later worked from.

1.3 Trust Boundary and Specification Reconciliation

The third phase identified the tax-exempt set – the addresses that bypass `BoardwalkToken`'s transfer tax – as the central trust boundary of each launch. The exempt set was traced through `LaunchFactory.createLaunch` and `BoardwalkToken.initialize` and enumerated explicitly. Each exempt address was then evaluated for whether its compromise would result in tax bypass and what structural defenses (notably the `RAISE_TOKEN` restriction in `BoardwalkLPManager`) prevent the exempt set from being used as a transfer tunnel.

In parallel, the specification was reconciled against the code. Each numerical bound, mathematical formula, and behavioral guarantee in the spec was located in code and either confirmed or marked as a deviation requiring follow-up. Where the spec described an explicit cap or rule (for example BPS ceilings, quorum percentages, timelock delays, win-streak limits, sequential finalization ordering, accounted-budget accounting), the corresponding code path was read and the reconciliation captured. Potential deviation scenarios and open questions were surfaced for the team during this phase.

1.4 Deep-Dive on Highest-Risk Areas

The final phase prioritized the highest-risk components for line-by-line review with explicit attack-path construction. Priority was set jointly with the team and was driven by three signals: novel mechanics with limited prior-art (the governance options, the multi-stage finalization pipeline), high blast-radius components (the per-launch atomic seed pipeline, the fee aggregation and migration path), and components with permissionless external entry points that interact with non-trivial external systems (the v4 LP minting and locking flow).

For each deep-dive area, the review:

1. Re-read the contract sources side-by-side with their interface counterparts and direct callers, tracing each external entry point to every storage write it can cause.
2. Constructed concrete adversarial scenarios – specifying caller, calldata, and externally-controlled state – rather than reasoning in the abstract.
3. Distinguished confirmed hypotheses from refuted ones, and from those that remained provisional pending external-source verification.
4. Recorded the assumptions each finding rested on, particularly assumptions about third-party code (Uniswap v4 PositionManager semantics, OpenZeppelin behavior, WETH semantics).
5. Drafted mitigations and ranked them by the size of the change required and the breadth of attack surface closed.

2. Techniques

Several recurring techniques structured the review.

Entry-point enumeration. Every external function across the codebase was catalogued and categorized as permissionless, role-gated, or self-authorized (where authorization comes from the function's identity logic rather than a single role). This catalogue was the basis for systematic attack-surface analysis: every permissionless function was treated as an adversary-controlled entry point until proven otherwise.

Trust-boundary tracing. Wherever code crossed a privilege boundary (for example, a permissionless function that ultimately writes privileged state, or a try/catch that swallows a failure on the privileged side) the boundary was made explicit and the conditions under which it could be crossed were enumerated.

State-machine reasoning for governance and timelock flows. The signal/cancel/execute pattern in `Timelocked`, the multi-epoch governance pipeline in `GovernanceVoter`, and the migration flow in `BoardwalkFeeCollector` were each modeled as state machines. Transitions were enumerated and tested for liveness and safety. Deadlock resolvers and force-execute paths were given particular attention.

Spec-to-code reconciliation. The published spec was used as a source of canonical numerical and behavioral claims. Each was located in code. Mismatches, even small ones, such as an implicit ceiling being looser than the stated cap, were recorded as deviations to confirm with the team.

Adversarial construction. The deep-dive phase constructed concrete attack scenarios with explicit caller, calldata, externally-controlled state, and resulting on-chain state changes. Each scenario was then stress-tested by attempting to refute it from the code, and any successful refutation was recorded alongside the original hypothesis.

External-dependency verification. Wherever the in-scope code depended on the semantics of an external contract (like the Uniswap v4 PositionManager), the dependency was identified, the specific semantic claim was isolated, and verification was attempted against the external source.

Issue M-1: First-vote lazy snapshot can be manipulated to lower governance quorum

Source: https://github.com/sherlock-audit/2026-04-bmx-apr-24th-2026/issues/19

Summary

`GovernanceVoter` is meant to require at least 51% voting participation before a governance option can take effect. The current implementation only records `snapshotTotalWeight` when the first vote of an epoch is cast, and it reads the live `sbfBMX.totalSupply()` at that moment.

This lets an attacker temporarily reduce part of their own `sbfBMX` supply before casting the first vote, lock in a smaller supply snapshot, restore the removed supply, and then continue voting from other addresses. Final quorum is then checked against the smaller snapshot, so the attacker can pass governance with less than 51% of the real voting supply.

Vulnerability Detail

The key issue is that the first voter can influence the quorum base for the whole epoch.

When `vote()` sees that the epoch has no snapshot yet, it calls `_primeEpoch(epoch)` during the first vote: GovernanceVoter.sol#L227-L228.

`_primeEpoch()` does not read an epoch-start checkpoint. It reads the external tracker's current `SBF_BMX.totalSupply()` directly: GovernanceVoter.sol#L501-L507. If the attacker can temporarily lower the live supply before the first vote, that lower value becomes the fixed snapshot for the epoch.

Vote weight is also read from the voter's live `SBF_BMX.balanceOf(msg.sender)` at vote time: GovernanceVoter.sol#L231-L252. So the attacker can lower the supply only long enough to set the snapshot, then restore the removed voting power and keep voting.

At finalization, quorum is calculated from `min(snapshotTotalWeight, liveSupply)`, and the required threshold is 51%: GovernanceVoter.sol#L465-L471. The 51% threshold is defined by `QUORUM_BPS = 5_100`: GovernanceVoter.sol#L27.

For example:

1. The real voting supply is `2000`, so the intended 51% quorum is `1020`.
2. The attacker controls `1000`, which should not be enough.
3. Before the first vote, the attacker temporarily removes `500`, making live `totalSupply()` equal to `1500`.
4. The first vote stores `snapshotTotalWeight = 1500`.

5. The attacker restores the removed `500` and votes with the rest of their controlled addresses.
6. Final quorum becomes `1500 * 51% = 765`, so the attacker's `1000` votes are enough.

The attacker does not control a majority of the real supply, but still makes their chosen option pass.

Impact

An attacker can lower the governance participation threshold and make an option pass without the intended quorum.

This contract controls the 70% governance share of protocol revenue: GovernanceVoter.sol#L18. Once quorum is suppressed, the attacker can redirect that epoch's governance-controlled revenue to a preferred route such as treasury, BuyBurnBMX, BuyBurnLP, or participation rewards.

Code Snippet

- The first vote initializes the epoch snapshot: GovernanceVoter.sol#L227-L228
- Vote weight uses the live balance at vote time: GovernanceVoter.sol#L231-L252
- The snapshot reads live `SBF_BMX.totalSupply()`: GovernanceVoter.sol#L501-L507
- Quorum uses the reduced snapshot as part of its base: GovernanceVoter.sol#L465-L471

Tool Used

Manual Review

Recommendation

Do not let the first vote decide the epoch's total voting-power snapshot. The quorum base should come from a checkpoint that cannot be manipulated by the first voter.

Recommended changes:

1. Snapshot `sbfBMX` total supply at the epoch boundary, not during the first vote.
2. Always calculate quorum from that fixed snapshot.
3. If the reward tracker cannot provide historical total supply, add a checkpoint mechanism or require a trusted keeper to initialize the epoch snapshot before voting starts.

Issue M-2: Cooldown can retroactively skip majority votes and let a minority option win

Source: https://github.com/sherlock-audit/2026-04-bmx-apr-24th-2026/issues/20

Summary

`GovernanceVoter` uses advance voting: votes in the current epoch decide the result for the next execution epoch. However, option eligibility is evaluated under different epochs at vote time and at winner selection time.

This can cause a majority to vote for an option that is eligible when they vote, but that option can become ineligible before execution because it triggers cooldown after consecutive wins. The majority votes no longer can win, but they still count toward quorum, allowing a minority-supported option to take the whole governance budget.

Vulnerability Detail

When a user votes, `vote()` checks whether the option is eligible in the current vote epoch, then records the vote weight: GovernanceVoter.sol#L216-L243.

At finalization, `_determineWinner()` first calculates quorum from the full `totalVoteWeight` of the prior vote epoch: GovernanceVoter.sol#L463-L471. It then selects the winner by checking `_isOptionEligible()` against the execution epoch being finalized: GovernanceVoter.sol#L483-L488.

Cooldown from consecutive wins is only updated after the current execution epoch finishes. `_updateConsecutiveWins()` marks the option ineligible for the next epoch after it reaches the maximum consecutive win count: GovernanceVoter.sol#L552-L563.

This creates a timing mismatch:

1. Option `X` has already won twice in a row and will enter cooldown if it wins once more.
2. Live vote epoch `N` opens while `X` is still eligible, and the majority votes for `X`.
3. A minority votes for another eligible option `Y`.
4. Later, when the current execution epoch is finalized, `X` gets its third consecutive win and becomes ineligible for epoch `N + 1`.
5. When epoch `N + 1` is finalized, the majority votes for `X` still count toward quorum.
6. But winner selection skips `X`, so the minority-supported `Y` becomes the winner.

The majority votes help satisfy quorum but cannot decide the winner, which breaks the intended governance result.

Impact

A minority voter can win the full governance budget for an epoch when the majority option is retroactively skipped by cooldown.

That budget corresponds to the 70% governance share of protocol revenue: GovernanceVoter.sol#L18. Funds can be routed to a minority-selected treasury, buy-and-burn, LP buy, or participation rewards path instead of the option supported by most voters.

Code Snippet

- Vote time checks eligibility against the current vote epoch: GovernanceVoter.sol#L216-L243
- Quorum uses the full total votes from the prior vote epoch: GovernanceVoter.sol#L463-L471
- Winner selection checks eligibility against the execution epoch: GovernanceVoter.sol#L483-L488
- Cooldown is updated to the next epoch only after execution: GovernanceVoter.sol#L552-L563

Tool Used

Manual Review

Recommendation

Voting and winner selection must use the same eligibility view.

Possible fixes:

1. Snapshot the target execution epoch's eligibility at the start of the vote epoch, then use that fixed result both during voting and finalization.
2. If an option is ineligible at finalization, votes for that option should not continue to count toward quorum.
3. Adjust when cooldown takes effect so already-open vote epochs are not changed retroactively.
4. Make the target execution epoch explicit in both the UI and contract logic, so users cannot vote for an option that will later be skipped by system rules.

Issue M-3: Option 3 double-encodes the V4 Position-Manager calldata before passing it to Universal Router

Source: https://github.com/sherlock-audit/2026-04-bmx-apr-24th-2026/issues/31

Summary

`GovernanceVoter._executeBuyBurnLp()` builds a valid `PositionManager.modifyLiquidities()` calldata blob, but then wraps it again with `abi.encode(pmCalldata)` before passing it as the input for Universal Router command `V4_POSITION_MANAGER_CALL`.

The Base Universal Router expects the command input to be the raw PositionManager calldata. It validates the first 4 bytes of the input as the `modifyLiquidities(bytes,uint256)` selector and then forwards the same bytes directly to the PositionManager. Because `abi.encode(pmCalldata)` starts with an ABI offset word instead of the selector, the router reads `0x00000000` as the action selector and reverts before any LP minting logic runs.

Vulnerability Detail

The Option 3 execution path constructs the PositionManager call:

```
bytes memory pmCalldata = abi.encodeCall(
    IV4PositionManager.modifyLiquidities, (abi.encode(mintActions, mintParams),
     ↪  deadline)
);
```

This `pmCalldata` already has the correct calldata layout:

```
modifyLiquidities(bytes,uint256) selector || encoded arguments
```

However, the code then stores `abi.encode(pmCalldata)` as the Universal Router input:

```
bytes memory commands = abi.encodePacked(UR_V4_POSITION_MANAGER_CALL);
bytes[] memory inputs = new bytes[](1);
inputs[0] = abi.encode(pmCalldata);
```

For the Base Universal Router, command `0x14` (`V4_POSITION_MANAGER_CALL`) expects `inputs[0]` to be the raw calldata to forward to the configured V4 PositionManager. Its dispatcher performs a safety check before forwarding:

```
_checkV4PositionManagerCall(inputs);
(success, output) = address(V4_POSITION_MANAGER).call{value:
 ↪  address(this).balance}(inputs);
```

The safety check reads the first 4 bytes of `inputs` and requires them to equal `V4_POSITION_MANAGER.modifyLiquidities.selector`. With the current double-encoding, the first 32 bytes of `inputs[0]` are the ABI offset for the dynamic bytes value, so the first 4 bytes are `0x00000000`, not the PositionManager selector.

On the real Base Universal Router, an `eth_call` using the current `GovernanceVoter` encoding reverts with:

```
InvalidAction(0x00000000)
```

As a result, the call never reaches `PositionManager.modifyLiquidities()` and the LP position cannot be minted.

Impact

Option 3 (`BuyBurnLP`) is non-executable with the default Base Universal Router integration.

If Option 3 wins governance, keeper/owner execution will revert before the LP mint. The epoch remains finalized but unexecuted until the forced fallback path becomes available, after which the budget can be routed to `fallbackTreasury` instead of being used to buy BMX and create a locked ETH/BMX LP position.

Code Snippet

https://github.com/sherlock-audit/2026-04-bmx-apr-24th-2026/blob/main/boardwalk-contracts/src/governance/GovernanceVoter.sol#L658-L666

Base Universal Router verified source:

https://basescan.org/address/0x6ff5693b99212da76ad316178a184ab56d299b43#code

Tool Used

Manual Review

Recommendation

Pass the raw PositionManager calldata as the Universal Router input:

```
inputs[0] = pmCalldata;
```

Do not wrap it with `abi.encode()`. Add a Base fork test for Option 3 that asserts the router accepts the `V4_POSITION_MANAGER_CALL` input and reaches the PositionManager call path.

Issue M-4: Option 3 sends BMX to Universal Router but PositionManager settles ERC20 through Permit2

Source: https://github.com/sherlock-audit/2026-04-bmx-apr-24th-2026/issues/32

Summary

`GovernanceVoter._executeBuyBurnLp()` transfers the BMX side of the LP mint to the Universal Router and then calls the V4 PositionManager through Universal Router command `V4_POSITION_MANAGER_CALL`.

This is not sufficient for the canonical Base PositionManager. During `SETTLE_PAIR`, the PositionManager settles owed ERC20 by calling Permit2 `transferFrom(payer, poolManager, ...)`, where `payer` is `msgSender()`. When the PositionManager is called through the Universal Router, `msgSender()` is the Universal Router, not the `GovernanceVoter`. Therefore the PositionManager attempts to pull BMX from the Universal Router through Permit2. The Universal Router has no BMX Permit2 allowance for the PositionManager, so settlement reverts even if the Router already holds enough BMX.

Vulnerability Detail

Option 3 first swaps half of the epoch budget into BMX and leaves the received BMX in `GovernanceVoter`:

```
uint256 bmxReceived = _swapRaiseTokenForBmx(halfForBmx, amountOutMin,
↪  address(this), deadline);
```

It then transfers that BMX to the Universal Router before asking the router to call the V4 PositionManager:

```
IERC20(BMX).safeTransfer(UNIVERSAL_ROUTER, bmxReceived);
...
IUniversalRouter(UNIVERSAL_ROUTER).execute{value: halfForEth}(commands, inputs,
↪  deadline);
```

The PositionManager action sequence is:

```
MINT_POSITION + SETTLE_PAIR + SWEEP
```

On the canonical Base PositionManager, `SETTLE_PAIR` uses the current `msgSender()` as the payer:

```
function _settlePair(Currency currency0, Currency currency1) internal {
    address caller = msgSender();
    _settle(currency0, caller, _getFullDebt(currency0));
```

```
    _settle(currency1, caller, _getFullDebt(currency1));
}
```

When called through the Universal Router, the PositionManager's reentrancy lock records the Universal Router as the locker, so `msgSender()` returns the Universal Router. For the native ETH side, `_settle()` can pay with `msg.value`. For the BMX ERC20 side, `_pay()` does not use the Universal Router's ordinary token balance unless `payer == address(this)`:

```
function _pay(Currency currency, address payer, uint256 amount) internal override {
    if (payer == address(this)) {
        currency.transfer(address(poolManager), amount);
    } else {
        permit2.transferFrom(payer, address(poolManager), uint160(amount),
         ↪  Currency.unwrap(currency));
    }
}
```

Here, `payer` is the Universal Router and `address(this)` is the PositionManager, so the `else` branch is used. The PositionManager attempts to pull BMX from the Universal Router through Permit2. Holding BMX directly in the Universal Router is not enough for this path; Permit2 must have both token approval and packed allowance for the PositionManager spender.

On Base, the default Universal Router currently has no such BMX allowance:

```
BMX allowance(UniversalRouter, Permit2) = 0
Permit2 allowance(UniversalRouter, BMX, PositionManager) = (0, 0, 0)
```

After correcting the earlier raw calldata encoding and using tick-aligned bounds, a Base `eth_call` reaches this settlement path and reverts with Permit2 `AllowanceExpired(0)`.

Impact

Option 3 has an additional execution blocker on the canonical Base V4 PositionManager integration.

Even if the Universal Router calldata encoding is fixed and the LP ticks are made valid for the configured tick spacing, the LP mint cannot settle the BMX side by merely transferring BMX to the Universal Router. Option 3 will still revert during ERC20 settlement, preventing the intended buy-and-LP governance outcome. The epoch can remain unexecuted until the forced fallback delay passes and the budget is routed to `fallbackTreasury`.

Code Snippet

https://github.com/sherlock-audit/2026-04-bmx-apr-24th-2026/blob/main/boardwalk-contracts/src/governance/GovernanceVoter.sol#L632-L666

Base PositionManager verified source:

https://basescan.org/address/0x7c5f5a4bbd8fd63184577525326123b519429bdc#code

Base Universal Router verified source:

https://basescan.org/address/0x6ff5693b99212da76ad316178a184ab56d299b43#code

Tool Used

Manual Review

Recommendation

Do not rely on the Universal Router's ordinary BMX balance for PositionManager `SETTLE_PAIR`.

Use a settlement path where the PositionManager pays from its own token balance, or route the mint through a proper Permit2 approval flow for the actual payer expected by the PositionManager. If the intended payer is `GovernanceVoter`, call the PositionManager directly with the required approvals instead of forwarding through the Universal Router. If the intended payer is the PositionManager itself, transfer BMX to the PositionManager and use an action path that settles from `address(this)` rather than `msgSender()`.

Add a Base fork test for Option 3 that reaches the ERC20 settlement step and verifies the BMX side is actually paid into the PoolManager.

Issue M-5: Fee recipient role rotation accepts arbitrary targets that can brick transfers, waste user gas, or disable AMM tax

Source: https://github.com/sherlock-audit/2026-04-bmx-apr-24th-2026/issues/33

Summary

`FeeDistributor.executeChangeIntegratorAddress()` and `executeChangeAncillaryAddress()` automatically grant tax exemption to the new role address, but only check that the address is non-zero and not already exempt.

There is no requirement that the new integrator/ancillary address is a `FeeRecipientCollector` or any other expected collector contract. A malicious or compromised current role holder can rotate to an EOA, a custom malicious collector, or the launch's AMM pair.

This creates three concrete attack paths from the same root cause: an EOA can brick subsequent taxed transfers, a malicious custom collector can force every taxed transfer to pay for unnecessary downstream execution, and an exempt AMM pair can disable all trading-tax revenue for that pair.

Vulnerability Detail

`BoardwalkToken._calculateTax()` returns zero if either transfer endpoint is exempt: BoardwalkToken.sol#L159-L180. This is not limited to protocol contracts. Any address in `isExempt` becomes a tax-free endpoint.

The current integrator can signal a new integrator address, and after the 14-day role timelock `executeChangeIntegratorAddress()` removes exemption from the old role and grants exemption to `newAddress`: FeeDistributor.sol#L336-L355. The only target validation is `newAddress != address(0)` and `!isExempt(newAddress)`.

Ancillary rotation has the same behavior: FeeDistributor.sol#L362-L381.

The intended batched role migration helper is `FeeRecipientCollector`, but its helper also accepts an arbitrary `newAddress` and forwards it to each distributor: FeeRecipientCollector.sol#L104-L139. The downstream distributor never verifies that `newAddress` is another collector, nor that it implements `notifyFees()`.

After any taxed transfer, `BoardwalkToken` calls `FeeDistributor.onTaxReceived()`. If the corresponding role BPS is positive, the distributor forwards the integrator/ancillary share on every taxed transfer: FeeDistributor.sol#L188-L204. `_forwardThirdParty()` first transfers the role share to the target, then calls `notifyFees` with a 150,000 gas cap: FeeDistributor.sol#L23-L26, FeeDistributor.sol#L441-L449.

Attack path 1: EOA/non-contract role target bricks taxed transfers.

1. The current integrator or ancillary role holder chooses an EOA or arbitrary contract as `newAddress`.
2. The role holder signals the matching role change.
3. After the 14-day timelock, anyone executes the pending role change.
4. `FeeDistributor` calls `BoardwalkToken.updateExempt(newAddress, true)` and sets `integrator` or `ancillary` to that address.
5. On the next ordinary taxed transfer, `onTaxReceived()` attempts to forward the positive role share.
6. `_forwardThirdParty()` transfers the role share and then performs a typed `IFeeRecipientCollector(newAddress).notifyFees(...)` call.
7. If `newAddress` is an EOA/non-contract, the typed call's contract-existence check reverts, so the whole token transfer reverts.
8. The EOA itself is still exempt for its own transfers, but normal taxed transfers by other users can be bricked.

Attack path 2: custom collector makes users pay for arbitrary downstream work.

1. The current integrator or ancillary role holder rotates to a custom contract that implements `notifyFees`.
2. Every ordinary taxed transfer now triggers `_forwardThirdParty()` for that custom target.
3. The custom `notifyFees()` can consume the full 150,000 forwarded gas budget, for example by performing token swaps and transferring proceeds to its owner.
4. The taxed transfer caller pays for this extra execution cost every time they transfer.
5. If both integrator and ancillary are controlled by malicious custom collectors, the extra downstream execution can be charged twice per taxed transfer.

Attack path 3: AMM pair exemption disables trading tax for the pair.

The pair is created or fetched during `PresaleManager.seedLiquidity()`: PresaleManager.sol#L206-L219. It is not part of the launch token's initial exempt list. `LaunchFactory._buildExemptList()` only adds the presale manager, LP staking, Boardwalk LP manager, boardwalk fee collector, optional vesting stream, and optional integrator/ancillary addresses: LaunchFactory.sol#L431-L453.

When the AMM pair is made exempt:

1. sells transfer launch tokens from the user to the pair, so `to == pair`;
2. buys transfer launch tokens from the pair to the user, so `from == pair`;
3. `_calculateTax()` returns zero for both directions;
4. `FeeDistributor.onTaxReceived()` is no longer called for swaps through that pair.

The role can also become practically stuck after rotation to an EOA/arbitrary contract or AMM pair. Future role changes require `msg.sender == integrator` or `msg.sender == ancillary`, so rotation depends entirely on that new address being willing and able to call the role-change functions. The ability to rotate integrator/ancillary after launch does not appear necessary for normal launch operation, so this mutable arbitrary target adds risk without a clear post-launch need.

The same missing target validation exists in the factory-level integrator/ancillary rotation for future launches: LaunchFactory.sol#L613-L654. Once the global role is changed, future launches inherit that address as exempt through `_buildExemptList()`.

Impact

A malicious or compromised current integrator/ancillary role holder can create an unsafe role target after the 14-day timelock.

If the target is an EOA/non-contract and the role BPS is positive, subsequent taxed transfers can revert, effectively denying normal token transfers that should pay tax. If the target is a malicious collector contract, every taxed transfer can be forced to pay up to the forwarded notify gas budget for attacker-controlled downstream execution. If the target is the AMM pair, all buys and sells through that pair bypass both the base tax and the anti-whale tax.

The AMM-pair path causes direct loss of all tax revenue that should have been collected from those swaps, including issuer fees, referrer fees when configured, LP staking incentives, boardwalk fees, integrator fees, and ancillary fees.

This is a trusted-role-exceeds-authority issue rather than a permissionless attack. The affected role is intended to be a fee recipient with a delayed migration path, not an actor able to brick taxed transfers, impose arbitrary notify execution costs on users, or disable launch tokenomics for an AMM pair.

Code Snippet

- Integrator rotation grants exemption to arbitrary non-exempt `newAddress`: FeeDistributor.sol#L336-L355
- Ancillary rotation grants exemption to arbitrary non-exempt `newAddress`: FeeDistributor.sol#L362-L381
- `FeeRecipientCollector` forwards arbitrary `newAddress` values to distributors: FeeRecipientCollector.sol#L104-L139
- Taxed transfers forward integrator/ancillary shares on each tax callback: FeeDistributor.sol#L188-L204
- Third-party forwarding calls arbitrary `notifyFees()` code with a 150,000 gas cap: FeeDistributor.sol#L23-L26, FeeDistributor.sol#L441-L449
- Tax returns zero when either endpoint is exempt: BoardwalkToken.sol#L159-L180

- The launch AMM pair is created during liquidity seeding: PresaleManager.sol#L206-L219
- The initial exempt list does not include the AMM pair: LaunchFactory.sol#L431-L453
- Future launch role rotation has the same missing target validation: LaunchFactory.sol#L613-L654

Tool Used

Manual Review

Recommendation

Do not let fee recipient role rotation grant tax exemption to arbitrary addresses.

Recommended fixes:

1. Require new integrator/ancillary addresses to be approved collector contracts, or at minimum require them to satisfy an explicit collector interface / registry check before exemption is granted.
2. Separate fee recipient rotation from tax exemption. A recipient address should not automatically become exempt unless it is a known safe collector contract that truly needs exemption.
3. Reject EOAs and arbitrary contracts as role targets unless they are explicitly allowlisted by the protocol.
4. Avoid calling arbitrary recipient logic on every taxed transfer. If notification is required, use a pull-based model or restrict callbacks to trusted collector implementations with bounded behavior.
5. Reject launch trading infrastructure as integrator/ancillary targets before granting exemption. At minimum, reject the launch AMM pair, token, router, factory, fee distributor, presale manager, LP staking, and other protocol-critical addresses.
6. Derive the pair from the configured DEX factory and `raiseToken` inside `FeeDistributor` and reject `newAddress == pair` for per-launch rotations.
7. If integrator/ancillary rotation is not needed after launch, remove or permanently disable these post-launch role rotation functions.
8. Apply equivalent target validation to `LaunchFactory.executeChangeIntegratorAddress()` and `executeChangeAncillaryAddress()` so future launches cannot inherit unsafe targets.

Issue L-1: Participation rewards are rounded down and leave BMX dust permanently locked

Source: https://github.com/sherlock-audit/2026-04-bmx-apr-24th-2026/issues/15

Summary

`ParticipationDistributor.claimable()` calculates each voter's final allocation with integer division:

```
totalAllocation = s.totalBmx * uv.weight / s.totalWeight;
```

Because every user's allocation is rounded down independently, the sum of all claimable allocations can be smaller than `s.totalBmx`. The remaining BMX has no eligible claimant and `ParticipationDistributor` does not expose any sweep or rescue mechanism, so this dust remains permanently stuck in the contract.

Vulnerability Detail

When governance option 4 wins, `GovernanceVoter` swaps the epoch budget into BMX and transfers the full received amount into `ParticipationDistributor` through `createStream( )`:

```
uint256 bmxReceived = _swapRaiseTokenForBmx(raiseAmount, amountOutMin,
 ↪  address(this), deadline);
IERC20(BMX).forceApprove(participationDistributor, bmxReceived);
IParticipationDistributor(participationDistributor).createStream(epoch,
 ↪  bmxReceived);
```

`createStream()` stores that full BMX amount as the stream's `totalBmx`:

```
IERC20(BMX).safeTransferFrom(msg.sender, address(this), bmxAmount);

streams[epoch] =
    StreamInfo({totalBmx: bmxAmount, totalWeight: priorEpoch.totalVoteWeight,
     ↪  startTime: block.timestamp});
```

However, the per-user allocation is calculated by flooring each user's proportional share:

```
totalAllocation = s.totalBmx * uv.weight / s.totalWeight;
```

If the total BMX amount does not divide exactly across all voter weights, the fractional parts are discarded. For example, if `totalBmx = 100` and three voters have equal weight with `totalWeight = 3`, each user receives `100 * 1 / 3 = 33`, for a total distributed amount of `99`. The remaining `1` BMX is not assigned to any user.

The same issue occurs in most streams where `s.totalBmx * uv.weight` is not exactly divisible by `s.totalWeight`. Since `claim()` and `claimAll()` only transfer a caller's rounded-down `claimableAmount`, and the contract has no function to sweep excess BMX after a stream ends, the undistributed remainder accumulates in `ParticipationDistributor` forever.

Impact

This is a Low/Info accounting issue rather than a direct theft vector.

Every participation stream can leave a small amount of BMX undistributed. The amount per stream is bounded by rounding dust, but it can accumulate across many epochs. These tokens cannot be claimed by voters, recovered by governance, sent to treasury, or included in a later stream under the current implementation.

Code Snippet

https://github.com/sherlock-audit/2026-04-bmx-apr-24th-2026/blob/main/boardwalk-contracts/src/governance/ParticipationDistributor.sol#L104

Tool Used

Manual Review

Recommendation

Add an explicit dust handling path for each stream.

Issue L-2: IParticipationDistributor exposes the wrong return value for claimable rewards

Source: https://github.com/sherlock-audit/2026-04-bmx-apr-24th-2026/issues/16

Summary

`ParticipationDistributor.claimable()` returns two values: `totalAllocation` and `claimableAmount`.

However, `IParticipationDistributor.claimable()` declares only one `uint256` return value. Since Solidity function selectors do not include return types, calls made through the interface still reach the implementation, but callers expecting one return value may decode the first returned word, `totalAllocation`, as if it were the currently claimable amount.

Vulnerability Detail

The implementation documents and returns both the user's full stream allocation and the amount claimable at the current timestamp:

```
function claimable(
    uint256 epoch,
    address user
) public view returns (uint256 totalAllocation, uint256 claimableAmount)
```

The first return value is the user's total allocation for the stream. It does not account for vesting progress or prior claims. The second return value is the actual amount that can be claimed now.

The interface defines the same function with only one return value:

```
function claimable(
    uint256 epoch,
    address user
) external view returns (uint256);
```

Because return values are not part of the ABI function selector, an external contract using `IParticipationDistributor` will call the correct target function. But the returned data layout begins with `totalAllocation`, followed by `claimableAmount`. A caller decoding a single `uint256` can therefore treat `totalAllocation` as the claimable amount.

This is especially misleading after a user has already claimed part or all of their allocation. For example, after the stream is fully vested and the user has claimed everything, the implementation would return `(totalAllocation, 0)`, while a call through

the interface may expose only `totalAllocation`. Integrations can then report a positive claimable balance even though `claim()` would revert with `NothingToClaim()`.

Impact

This is a Low/Info integration correctness issue.

The core `ParticipationDistributor` claim flow uses the implementation directly and reads the second value internally. However, external contracts, indexers, dashboards, or helper contracts that rely on `IParticipationDistributor.claimable()` can display or consume the wrong value. This can lead to inaccurate reward accounting, failed claim attempts, and misleading user-facing balances.

Code Snippet

https://github.com/sherlock-audit/2026-04-bmx-apr-24th-2026/blob/main/boardwalk-contracts/src/governance/ParticipationDistributor.sol#L94-L97

https://github.com/sherlock-audit/2026-04-bmx-apr-24th-2026/blob/main/boardwalk-contracts/src/interfaces/IParticipationDistributor.sol#L27-L30

Tool Used

Manual Review

Recommendation

Update the interface so it exactly matches the implementation:

```
function claimable(
    uint256 epoch,
    address user
) external view returns (uint256 totalAllocation, uint256 claimableAmount);
```

Issue L-3: Option 3 uses hardcoded LP ticks that may be incompatible with the configured tick spacing

Source: https://github.com/sherlock-audit/2026-04-bmx-apr-24th-2026/issues/17

Summary

`GovernanceVoter` accepts `POOL_TICK_SPACING` as a constructor parameter, but the Option 3 execution path hardcodes the LP position range to `-887220` and `887220`.

Those ticks are compatible with common spacing values such as `1`, `10`, and `60`, but they are not guaranteed to be valid for every configurable spacing. If the contract is deployed with an incompatible spacing, the Uniswap V4 position mint can revert because the lower and upper ticks are not aligned to the pool's tick spacing.

Vulnerability Detail

The deployment parameters include a configurable `poolTickSpacing`, which is stored as `POOL_TICK_SPACING`.

Later, when Option 3 (`BuyBurnLP`) is executed, `_executeBuyBurnLp()` passes that configured spacing into the pool key, but still uses fixed full-range ticks:

- lower tick: `-887220`
- upper tick: `887220`

Uniswap V4 LP positions require the selected ticks to be usable for the pool's tick spacing. A tick is usable only when it is aligned to the spacing. While `887220` is divisible by common spacings such as `1`, `10`, and `60`, it is not divisible by every possible spacing that can be provided through the constructor.

For example, if `POOL_TICK_SPACING` is configured as `100` or `200`, the hardcoded ticks are not aligned to that spacing. In that case, Option 3 can win governance, but `execute()` will revert when trying to mint the LP position.

The contract already treats tick spacing as a deployment-time configuration, so the usable full-range ticks should be derived from that configuration rather than hardcoded for only some spacings.

Impact

If the chosen tick spacing is incompatible with the hardcoded ticks, Option 3 becomes non-executable.

When Option 3 wins, keeper/owner execution will keep reverting during the LP mint path. The epoch will remain finalized but unexecuted until the forced fallback delay passes, after which `forceMarkExecuted()` can route the budget to `fallbackTreasury` instead of

creating and locking the intended LP position. This breaks the governance result for that epoch and delays budget handling.

Code Snippet

https://github.com/sherlock-audit/2026-04-bmx-apr-24th-2026/blob/main/boardwalk-contracts/src/governance/GovernanceVoter.sol#L138-L152

https://github.com/sherlock-audit/2026-04-bmx-apr-24th-2026/blob/main/boardwalk-contracts/src/governance/GovernanceVoter.sol#L641-L649

Tool Used

Manual Review

Recommendation

Derive the LP position bounds from the configured spacing instead of hardcoding them.

Use Uniswap's TickMath helpers for the deployed pool spacing, for example:

```
int24 tickLower = TickMath.minUsableTick(POOL_TICK_SPACING);
int24 tickUpper = TickMath.maxUsableTick(POOL_TICK_SPACING);
```

Alternatively, if the protocol intentionally supports only a restricted set of tick spacings, validate `poolTickSpacing` in the constructor and reject any value for which the hardcoded ticks are not usable.

Issue L-4: Delayed finalization allows immediate forced fallback execution

Source: https://github.com/sherlock-audit/2026-04-bmx-apr-24th-2026/issues/18

Summary

`GovernanceVoter.forceMarkExecuted()` starts its forced-execution delay from the epoch end time, not from the time the epoch is finalized.

If a keeper/owner finalizes an epoch only after `epochEnd + FORCE_EXECUTE_DELAY` has already passed, the epoch becomes immediately eligible for `forceMarkExecuted()` as soon as finalization completes. Any external caller can then mark the epoch as executed and route the full budget to `fallbackTreasury`, preventing the actual winning governance option from being executed.

Vulnerability Detail

The normal lifecycle expects an epoch to be finalized first and then executed according to its `winningOption`. If execution gets stuck, `forceMarkExecuted()` is intended to act as a deadlock resolver after an additional delay.

However, the forced fallback delay is measured from the epoch's scheduled end:

```
uint256 epochEnd = EPOCH_ZERO + (epoch + 1) * EPOCH_DURATION;
if (block.timestamp < epochEnd + FORCE_EXECUTE_DELAY) revert EpochNotOverdue();
```

This ignores when the epoch was actually finalized. Since `finalize()` can be delayed, an epoch may be finalized long after the forced fallback deadline has already passed.

Example flow:

1. An epoch ends with a non-treasury winning option, such as `BuyBurnBMX`, `BuyBurnLP`, or `Participation`.
2. The keeper/owner does not call `finalize()` until more than `FORCE_EXECUTE_DELAY` has passed after the epoch end.
3. `finalize(epoch, maxBatch)` records `winningOption` and `budget`.
4. Because `block.timestamp >= epochEnd + FORCE_EXECUTE_DELAY` is already true, anyone can immediately call `forceMarkExecuted(epoch)`.
5. `forceMarkExecuted()` sets `e.executed = true`, subtracts the epoch budget from `accountedBudget`, and transfers the budget to `fallbackTreasury`.
6. The intended `execute()` path can no longer run because the epoch is already marked as executed.

As a result, a mechanism intended to recover from stuck execution can become an immediate bypass whenever finalization is delayed.

Impact

The winning governance result can be skipped for delayed epochs.

If the winning option is `BuyBurnBMX`, the protocol will not buy and burn BMX. If it is `BuyBurnLP`, the protocol will not create and lock LP. If it is `Participation`, voters will not receive the intended BMX stream. Instead, the full epoch budget is sent to `fallbackTreasury`.

The attacker does not need keeper privileges. They only need to call `forceMarkExecuted()` after a delayed finalization and before the keeper/owner executes the winning option.

Code Snippet

https://github.com/sherlock-audit/2026-04-bmx-apr-24th-2026/blob/main/boardwalk-contracts/src/governance/GovernanceVoter.sol#L262-L304

https://github.com/sherlock-audit/2026-04-bmx-apr-24th-2026/blob/main/boardwalk-contracts/src/governance/GovernanceVoter.sol#L306-L344

https://github.com/sherlock-audit/2026-04-bmx-apr-24th-2026/blob/main/boardwalk-contracts/src/governance/GovernanceVoter.sol#L346-L369

Tool Used

Manual Review

Recommendation

Start the forced fallback delay from the time the epoch is finalized, not from the epoch end time.

For example, record a `finalizedAt` timestamp when finalization completes:

```
mapping(uint256 => uint256) public finalizedAt;
```

Then set it in `finalize()`:

```
e.finalized = true;
finalizedAt[epoch] = block.timestamp;
```

Finally, update `forceMarkExecuted()` to use that timestamp:

```
if (block.timestamp < finalizedAt[epoch] + FORCE_EXECUTE_DELAY) revert
↪  EpochNotOverdue();
```

This gives every finalized epoch a full execution window before the forced fallback path becomes available, even if the keeper/owner finalized the epoch late.

Issue L-5: Option 3 can leave unused BMX inside Universal Router where it can be swept by others

Source: https://github.com/sherlock-audit/2026-04-bmx-apr-24th-2026/issues/21

Summary

Option 3 first swaps half of the budget into BMX, transfers all received BMX into `UNIVERSAL_ROUTER`, and then mints a v4 LP position.

The issue is that the LP mint's `amountMax` values are upper bounds, not guaranteed spend amounts. If the mint consumes less BMX than provided, the leftover BMX remains inside Universal Router. The current cleanup only handles BMX dust held by `GovernanceVoter` itself and does not clean router-held balances. A later caller can sweep those BMX tokens out of the router.

Vulnerability Detail

`_executeBuyBurnLp()` splits `raiseAmount` in half, swaps one half into BMX, keeps the other half as ETH, and transfers the full `bmxReceived` amount into `UNIVERSAL_ROUTER`: GovernanceVoter.sol#L624-L636.

The v4 actions are then encoded as `MINT_POSITION + SETTLE_PAIR + SWEEP`, but the final `SWEEP` only targets `address(0)`, meaning native ETH. It does not sweep BMX: GovernanceVoter.sol#L638-L666.

In the v4 PositionManager mint flow, `amount0Max` and `amount1Max` are maximum input bounds, not fixed amounts that must be fully consumed: PositionManager.sol#L213-L224, PositionManager.sol#L381-L384. The mint can legitimately use less BMX, or in extreme parameter choices, no BMX.

After execution, `GovernanceVoter` only checks and burns BMX dust held by its own address: GovernanceVoter.sol#L670-L671. If the BMX is sitting inside `UNIVERSAL_ROUTER`, this cleanup does not reach it.

Attack path:

1. A governance epoch resolves to option 3.
2. The keeper executes option 3, and the contract swaps half of the budget into BMX.
3. The contract transfers all BMX into Universal Router.
4. The LP mint consumes less BMX than `bmxReceived`.
5. The leftover BMX remains inside Universal Router.
6. A later attacker calls a router sweep command such as `SWEEP(BMX, attacker, 0)` and transfers the leftover BMX to themselves.

Impact

The protocol loses whatever part of the option 3 BMX budget was not consumed by the LP mint. In the worst case, if the mint consumes no BMX, the attacker can take the entire BMX half of the option 3 budget.

Code Snippet

- The full `bmxReceived` amount is transferred into Universal Router: GovernanceVoter.sol#L624-L636
- The actions sweep native ETH, not BMX: GovernanceVoter.sol#L638-L666
- The mint uses maximum input bounds: PositionManager.sol#L213-L224
- Local dust cleanup does not cover router-held BMX: GovernanceVoter.sol#L670-L671

Tool Used

Manual Review

Recommendation

Do not leave uncontrolled BMX balances inside Universal Router.

Recommended fixes:

1. Explicitly add a BMX sweep to the router actions, returning unused BMX to `GovernanceVoter` or burning it.
2. After execution, verify that `UNIVERSAL_ROUTER` does not retain BMX.
3. Provide the router only with the BMX amount actually needed, instead of transferring all `bmxReceived` upfront.
4. Add reasonable lower bounds for `liquidity` and amount parameters to avoid zero-liquidity or obviously under-consuming executions.

Issue L-6: Batched finalization globally blocks normal voting in the live epoch

Source: https://github.com/sherlock-audit/2026-04-bmx-apr-24th-2026/issues/22

Summary

`GovernanceVoter.vote()` reverts whenever `finalizationInProgress == true`. That flag is global and does not distinguish between the historical epoch being finalized and the current live epoch in which users want to vote.

When the previous voter set is large, finalization must be completed across multiple transactions. From the first batch until the last batch finishes, all current-epoch votes are blocked, causing users to lose voting time.

Vulnerability Detail

`vote()` checks the global `finalizationInProgress` flag at the start and reverts if it is true: GovernanceVoter.sol#L216-L220.

`finalize()` sets `finalizationInProgress = true` before validating historical voters. If the current `maxBatch` does not process all voters, the function returns early and waits for a later finalize call to continue: GovernanceVoter.sol#L272-L293.

`_validateVoters()` is explicitly batch-limited by `maxBatch`: GovernanceVoter.sol#L513-L541. Therefore, a large historical voter set naturally puts the contract into a multi-transaction finalization state.

Failure path:

1. An attacker or large group splits votes across many addresses in vote epoch `N`.
2. A keeper later finalizes execution epoch `N + 1`, which requires batched validation of epoch `N` voters.
3. The first finalize batch sets `finalizationInProgress = true` and returns before all voters are processed.
4. Before the remaining batches finish, ordinary users try to vote in the current live epoch.
5. `vote()` reverts due to the global flag.
6. If the voter set is large or the keeper is slow to complete follow-up batches, a large part of the voting window can be lost.

Impact

This is a governance liveness issue. Users cannot participate in the current epoch while the contract is still processing historical finalization batches. An attacker can increase the number of batches by creating many voters, compressing or blocking later vote windows.

Code Snippet

- `vote()` is blocked by the global `finalizationInProgress` flag: GovernanceVoter.sol#L216-L220
- `finalize()` sets the flag and returns early when the batch is incomplete: GovernanceVoter.sol#L272-L293
- `_validateVoters()` processes at most `maxBatch` voters: GovernanceVoter.sol#L513-L541

Tool Used

Manual Review

Recommendation

Do not block all voting with a single global finalization flag.

Recommended fixes:

1. Track finalization state per epoch and only restrict operations that directly conflict with the epoch being processed.
2. Allow voting in the current live epoch while a historical epoch is being finalized in batches.
3. If state protection is needed, use a narrower lock instead of globally disabling `vote()`.
4. Add clearer limits or incentives for voter-count and batch processing so finalization cannot remain suspended for a long time.

Issue L-7: Any user can inject unrelated NFTs into `LPLocker` and break batched fee claims

Source: https://github.com/sherlock-audit/2026-04-bmx-apr-24th-2026/issues/24

Summary

`LPLocker` only checks that an incoming NFT comes from the configured Uniswap v4 PositionManager. It does not check whether the NFT was created by `GovernanceVoter`, and it does not check whether the NFT belongs to the expected pool.

An attacker can therefore transfer any PositionManager NFT into `LPLocker`. That unrelated NFT is added to `lockedTokenIds`. Later, `claimAllFees()` treats it as a valid locked governance position, and the entire batch can revert.

Vulnerability Detail

`onERC721Received()` performs a weak check. It only requires `msg.sender == POSITION_MANAGER`, then immediately appends the `tokenId` to `lockedTokenIds`: LPLocker.sol#L54-L62.

This means any user who owns a real PositionManager NFT can call `safeTransferFrom()` on the PositionManager and send their NFT to `LPLocker`. `LPLocker` cannot distinguish whether the NFT came from the intended governance flow.

After that, `claimAllFees()` loops through every stored `lockedTokenIds` entry and does not isolate failures for individual tokens: LPLocker.sol#L74-L82.

`_claimFees()` then calls PositionManager's `modifyLiquidities()` using the locker's fixed `CURRENCY0`/`CURRENCY1` parameters: LPLocker.sol#L100-L112. If the injected NFT belongs to another pool, or otherwise triggers an error, the whole batched claim fails.

Attack path:

1. The attacker owns any NFT from the configured PositionManager.
2. The attacker transfers it to `LPLocker` using `safeTransferFrom()`.
3. `LPLocker` accepts the NFT and stores the `tokenId`.
4. Someone later calls `claimAllFees()`.
5. The locker tries to process the injected NFT using its fixed pool parameters.
6. The external call reverts, causing the whole batched fee claim to fail.

Impact

The batched fee-claim path for governance LP positions can be griefed or DoSed. The more unrelated NFTs an attacker injects, the more expensive and complex operational

recovery becomes, and protocol fee collection from locked LP positions can be delayed or blocked.

Code Snippet

- `onERC721Received()` accepts any NFT from the PositionManager: LPLocker.sol#L54-L62
- `claimAllFees()` loops directly over all token IDs: LPLocker.sol#L74-L82
- `_claimFees()` assumes every token ID belongs to the fixed pool: LPLocker.sol#L100-L112

Tool Used

Manual Review

Recommendation

`LPLocker` should not accept arbitrary externally sent PositionManager NFTs.

Recommended fixes:

1. Only accept token IDs that were initiated by `GovernanceVoter` and are in a pending state.
2. Validate the position's pool, currencies, fee, tick spacing, and other parameters when receiving the NFT.
3. Add a controlled removal path for mistakenly received or maliciously injected NFTs.
4. Make `claimAllFees()` isolate failures per token, so one bad token cannot block the entire batch.

Issue L-8: Fee collector migration can strand already-collected launch-token fees in the old collector

Source: https://github.com/sherlock-audit/2026-04-bmx-apr-24th-2026/issues/25

Summary

`BoardwalkFeeCollector` can migrate to a new collector. During migration, `FeeDistributor.setFeeCollector()` removes the old collector from the launch token's exempt list and grants exemption to the new collector.

If the old collector already holds unswapped launch-token fees, those balances lose their exempt status after migration. The old collector's `swapToRaiseToken()` relies on the standard router path and assumes token transfers are not fee-on-transfer. Once the old collector is no longer exempt, the swap can permanently revert and leave already-collected fees stuck in the old collector.

Vulnerability Detail

`BoardwalkFeeCollector.swapToRaiseToken()` uses standard `swapExactTokensForTokens()` to convert collected launch tokens into the raise token: BoardwalkFeeCollector.sol#L78-L119. This path relies on the collector being tax-exempt; otherwise, transferring launch tokens into the pair becomes fee-on-transfer behavior and breaks the standard router's input assumptions.

`executeMigrateCollector()` only tells downstream distributors to update their collector. It does not first empty token balances already held by the old collector: BoardwalkFeeCollector.sol#L171-L187.

`FeeDistributor.setFeeCollector()` immediately disables exemption for the old collector and enables exemption for the new collector: FeeDistributor.sol#L333-L342.

In `BoardwalkToken`, only the fee distributor can update exemptions, and non-exempt launch-token transfers enter the normal tax logic: BoardwalkToken.sol#L110-L146. The old collector cannot re-authorize itself later.

Problem path:

1. Some launch-token fees have already been forwarded into the current `BoardwalkFeeCollector`.
2. Those fees have not yet been swapped into the raise token.
3. The owner executes collector migration.
4. The distributor removes the old collector from the exempt list.
5. A keeper later tries to make the old collector swap the remaining launch tokens into the raise token.

6. The old collector is no longer exempt, so the standard router swap hits fee-on-transfer behavior and fails.
7. There is no sweep or rescue method, so the balance can remain permanently stuck in the old collector.

Impact

Already-collected and accounted boardwalk fees can become stranded during a valid migration. They cannot be converted into the raise token or sent to treasury / governance vault. The larger the old collector's unswapped balance at migration time, the larger the protocol asset loss.

Code Snippet

- The old collector uses standard router swaps: BoardwalkFeeCollector.sol#L78-L119
- Migration does not clear the old collector's balances: BoardwalkFeeCollector.sol#L171-L187
- `setFeeCollector()` removes the old collector exemption: FeeDistributor.sol#L333-L342
- Non-exempt launch-token transfers are taxed: BoardwalkToken.sol#L110-L146

Tool Used

Manual Review

Recommendation

Collector migration should be balance-safe.

Recommended fixes:

1. Require the old collector to clear all launch-token balances before migration.
2. Or transfer the old collector's balances safely to the new collector during migration before removing the old exemption.
3. Add a controlled rescue or sweep method in the old collector to move pre-migration balances to the new collector or treasury.
4. If the old collector may still need to process historical balances, keep temporary exemption until those balances are confirmed empty.

Issue L-9: LP fees accrued during zero-staker periods can be captured by the next staker

Source: https://github.com/sherlock-audit/2026-04-bmx-apr-24th-2026/issues/26

Summary

The design states that when `totalWeight == 0`, vesting and fee rewards for that period should be permanently lost to avoid unbounded reward carryover while nobody is staking.

However, `LPStaking` still accumulates fees into `pendingEpochFees` during zero-staker periods. After the epoch expires, the first future staker triggers epoch rollover, and the old `pendingEpochFees` are promoted into `currentEpochFees`. Those fees are then streamed to the current staker, allowing the next staker to claim historical fees they did not earn.

Vulnerability Detail

`notifyFees()` adds fees to `pendingEpochFees` without requiring any current stakers: LPStaking.sol#L164-L174.

When the epoch expires, `_advanceEpochIfNeeded()` unconditionally promotes `pendingEpochFees` into `currentEpochFees` and then clears `pendingEpochFees`: LPStaking.sol#L205-L215.

`_distributeRewards()` says zero-staker rewards are permanently lost, and the code does only update `lastRewardUpdate` and return when `totalWeight == 0`: LPStaking.sol#L218-L227. But that only affects rewards currently being distributed. It does not discard historical fees still sitting in `pendingEpochFees`.

Then `_calculateFeeReward()` streams `currentEpochFees` over the new epoch to whoever is currently staked: LPStaking.sol#L256-L269.

This contradicts the SPEC statement that rewards during zero-staker periods are permanently lost: SPEC.md#L177.

Attack path:

1. No LP is currently staked, so `totalWeight == 0`.
2. The launch token continues generating tax fees, and `FeeDistributor` calls `notifyFees()`, adding fees to `pendingEpochFees`.
3. The whole epoch passes with no one staking.
4. The attacker becomes the first staker after the epoch expires.
5. The attacker's `stake()` call triggers `_advanceEpochIfNeeded()`, promoting historical `pendingEpochFees` into the new `currentEpochFees`.

6. Those historical fees are then distributed to the attacker instead of being permanently lost.

Impact

An opportunistic staker can capture up to nearly 100% of the LP fee bucket accrued during a zero-staker period. In the simplest case, if the attacker is the only staker after rollover, they can claim the entire historical fee bucket that the design intended to discard.

Code Snippet

- `notifyFees()` adds to `pendingEpochFees` even when nobody is staking: LPStaking.sol#L164-L174
- Epoch rollover promotes pending fees into current fees: LPStaking.sol#L205-L215
- The zero-staker branch does not discard `pendingEpochFees`: LPStaking.sol#L218-L227
- The new epoch streams `currentEpochFees` to current stakers: LPStaking.sol#L256-L269

Tool Used

Manual Review

Recommendation

When a zero-staker epoch expires, the related `pendingEpochFees` should be explicitly discarded instead of carried into the next staker set.

Recommended fixes:

1. In `_advanceEpochIfNeeded()`, if the expired epoch had no stakers, clear `pendingEpochFees` or send it to a protocol-defined sink instead of assigning it to `currentEpochFees`.
2. Track which accrual epoch each fee bucket belongs to, and only allow users with staking weight in that epoch to receive it.
3. Add tests covering this case: after `notifyFees()` while nobody is staked, staking after epoch expiry must not let the new staker claim historical fees.

Issue L-10: Delayed finalization can attribute later governance revenue to an older epoch

Source: https://github.com/sherlock-audit/2026-04-bmx-apr-24th-2026/issues/29

Summary

`GovernanceVoter.finalize()` calculates an epoch's budget from the contract's current WETH balance minus `accountedBudget`.

The contract does not record which epoch the governance revenue belonged to when the revenue entered the vault. If an old epoch is finalized late, WETH received during later epochs can be assigned to that old epoch and then executed according to the old epoch's winning option.

Vulnerability Detail

Governance revenue is transferred into `GovernanceVoter` by `BoardwalkFeeCollector` after fees are swapped into the raise token. The transfer is a plain ERC20 transfer to `governanceVault`; it does not call a dedicated accounting entrypoint that records the target epoch.

Instead, budget assignment happens only when `finalize()` is called. On the first finalization step for an epoch, the contract reads its current WETH balance and assigns all unaccounted balance to the epoch being finalized:

- `currentBalance` is read from `IERC20(WETH).balanceOf(address(this))`
- `e.budget` is set to `currentBalance - accountedBudget`
- `accountedBudget` is updated to `currentBalance`

This means budget attribution depends on when the keeper/owner finalizes, not when revenue actually entered the governance vault.

Example flow:

1. Epoch 10 is the oldest unfinalized epoch.
2. Epoch 10 ends, but the keeper/owner does not finalize it.
3. During epochs 11, 12, and 13, additional governance revenue is transferred into `GovernanceVoter`.
4. The keeper/owner later finalizes epoch 10.
5. `finalize(10, maxBatch)` assigns the full unaccounted current WETH balance to epoch 10, including revenue received during later epochs.
6. `execute(10, ...)` then spends that full amount according to epoch 10's `winningOption`.

The same issue affects catch-up finalization. If several epochs are finalized after a delay, the first old epoch can absorb all accumulated unaccounted WETH, while later epochs may receive little or no budget even though revenue arrived during those later periods.

Impact

This is a Low severity accounting issue.

Revenue can be attributed to a different epoch than the one in which it was received. As a result, an older vote can determine the use of later revenue. For example, later revenue may be sent to treasury, used for buy/burn, used for LP creation, or sent to participation rewards based on a prior epoch's winner rather than the intended later governance cycle.

The issue mainly affects delayed keeper/owner operation or catch-up finalization. It does not directly steal funds, but it can make protocol revenue allocation depend on finalization timing instead of the intended epoch accounting.

Code Snippet

https://github.com/sherlock-audit/2026-04-bmx-apr-24th-2026/blob/main/boardwalk-contracts/src/governance/GovernanceVoter.sol#L262-L304

https://github.com/sherlock-audit/2026-04-bmx-apr-24th-2026/blob/main/boardwalk-contracts/src/governance/GovernanceVoter.sol#L306-L344

https://github.com/sherlock-audit/2026-04-bmx-apr-24th-2026/blob/main/boardwalk-contracts/src/core/BoardwalkFeeCollector.sol#L126-L135

Tool Used

Manual Review

Recommendation

Do not determine epoch budgets from the vault's current balance at finalization time.

Record governance revenue into per-epoch accounting when the revenue enters the governance vault. For example, route revenue through a controlled deposit function that transfers WETH and increments `epochRevenue[targetEpoch]`. Then `finalize(epoch)` should use the already-recorded amount for that epoch instead of `balanceOf(address(this)) - accountedBudget`.

If the intended design is advance voting, where epoch `N - 1` decides how epoch `N` revenue is used, the deposit path should explicitly map incoming revenue to the correct target budget epoch. This prevents keeper delays or catch-up finalization from changing which vote controls the revenue.

Issue L-11: Default Base Universal Router version differs from the local v4-periphery dependency

Source: https://github.com/sherlock-audit/2026-04-bmx-apr-24th-2026/issues/30

Summary

`GovernanceVoter` builds Uniswap v4 Universal Router calldata manually, while the deployment script does not deploy a router from the repository's local `lib/v4-periphery` checkout. Instead, the Base governance deployment script passes Uniswap's already-deployed Base Universal Router address by default.

This means calldata compatibility must be judged against the source code of the configured onchain router, not only against the local `foundry.lock` dependency. In this repository, the local `v4-periphery` revision includes the newer `minHopPriceX36` field in `ExactInputSingleParams`, but the default Base Universal Router source uses the older struct without that field and accepts the current `0x140` encoding.

Vulnerability Detail

`_swapRaiseTokenForBmx()` encodes the `SWAP_EXACT_IN_SINGLE` params as nine top-level fields:

```
swapParams[0] = abi.encode(
    address(0),
    BMX,
    POOL_FEE,
    POOL_TICK_SPACING,
    POOL_HOOKS,
    true,
    amountIn.toUint128(),
    amountOutMin.toUint128(),
    ""
);
```

This encoding is compatible with the deployed Base Universal Router version used by the default governance deployment script. The script defaults to:

- `BASE_UNIVERSAL_ROUTER = 0x6fF5693b99212Da76ad316178A184AB56D299b43`
- `BASE_V4_POSITION_MANAGER = 0x7C5f5A4bBd8fD63184577525326123B519429bDc`

Uniswap's official v4 deployments page lists the same addresses for Base. BaseScan verifies the router as `UniversalRouter`, compiled with Solidity `0.8.26` and optimizer `44444444` runs.

The verified source for that deployed router defines `ExactInputSingleParams` as:

```
struct ExactInputSingleParams {
    PoolKey poolKey;
    bool zeroForOne;
    uint128 amountIn;
    uint128 amountOutMinimum;
    bytes hookData;
}
```

Its `decodeSwapExactInSingleParams()` also checks for a minimum length of `0x140`, not `0x160`:

```
// 0x140 = 10 * 0x20 -> 8 elements, bytes offset, and bytes length 0
if lt(params.length, 0x140) {
    mstore(0, SLICE_ERROR_SELECTOR)
    revert(0x1c, 4)
}
```

By contrast, the repository's local `foundry.lock` pins `lib/v4-periphery` to revision `686f621d9b675fc78bf02781f59ec1ad36921706`, where `ExactInputSingleParams` includes an additional `minHopPriceX36` field and the decoder minimum length is `0x160`.

Therefore, using the local dependency alone to assess the script's default Base deployment can produce a false positive: the default onchain router accepts the current `0x140` swap parameter encoding. The compatibility issue only becomes relevant if a deployer overrides `UNIVERSAL_ROUTER` with a newer router whose v4-periphery dependency expects `minHopPriceX36`.

Impact

This is an Informational deployment and audit-assumption issue.

Under the script's default Base deployment parameters, the current GovernanceVoter swap encoding is compatible with the configured onchain Universal Router, so the claimed `minHopPriceX36`-based swap DoS is not triggered.

The practical risk is that reviewers, tests, or future deployment operators may treat the local `v4-periphery` checkout as authoritative for the already-deployed router address. That can lead to incorrect vulnerability reports, incorrect fixes, or unsafe router overrides. If the router address is overridden to a newer implementation without updating the calldata encoding, the governance swap paths can become incompatible.

Code Snippet

https://github.com/sherlock-audit/2026-04-bmx-apr-24th-2026/blob/main/boardwalk-contracts/script/03_DeployGovernance.s.sol#L21-L35

https://github.com/sherlock-audit/2026-04-bmx-apr-24th-2026/blob/main/boardwalk-contracts/src/governance/GovernanceVoter.sol#L590-L600

https://github.com/sherlock-audit/2026-04-bmx-apr-24th-2026/blob/main/boardwalk-contracts/foundry.lock#L14-L16

Uniswap Base v4 deployments:

https://developers.uniswap.org/docs/protocols/v4/deployments

Base Universal Router verified source:

https://basescan.org/address/0x6ff5693b99212da76ad316178a184ab56d299b43#code

Tool Used

Manual Review

Recommendation

Document that the governance deployment script is compatible with the specific Base Universal Router version at `0x6fF5693b99212Da76ad316178A184AB56D299b43`, not with every `v4-periphery` revision.

Add a Base fork compatibility test that builds the exact `_swapRaiseTokenForBmx()` `SWAP_EXACT_IN_SINGLE` calldata and verifies that the default router accepts the `0x140` parameter layout.

If `UNIVERSAL_ROUTER` remains overrideable through the deployment environment, require an explicit compatibility check for the target router version. A deployment using a newer router that expects `minHopPriceX36` should either use a different encoding path or be rejected before deployment.

Issue L-12: Advanced launches with a 50% presale can deploy a permanently uninitialized VestingStream

Source: https://github.com/sherlock-audit/2026-04-bmx-apr-24th-2026/issues/35

Summary

Advanced launches with `presalePercent == 5000` do not have an issuer vesting allocation because the full supply is split between presale tokens and liquidity tokens. However, `LaunchFactory.createLaunch()` still deploys and records a `VestingStream` whenever the issuer provides non-empty vesting recipients.

During liquidity seeding, `PresaleManager.seedLiquidity()` only mints issuer vesting tokens and initializes the vesting stream when `issuerVestingTokens > 0`. For a 50% presale this value is zero, so the deployed `VestingStream` remains uninitialized, receives no tokens, and cannot be used by the configured recipients.

Vulnerability Detail

The specification states that `VestingStream` is used for the Advanced path only when `presalePercent < 50%`, and that Advanced launches deploy 5 clones only in that case. Otherwise they should deploy 4 clones.

The implementation does not enforce that boundary. `LaunchFactory.createLaunch()` deploys a vesting stream based only on whether `config.vestingRecipients.length > 0`:

```
address vestingAddr;
if (config.vestingRecipients.length > 0) {
    vestingAddr = Clones.clone(VESTING_IMPL);
}
```

For Advanced launches, validation requires vesting recipients when `presalePercent < 5000`, but it does not reject vesting recipients when `presalePercent == 5000`:

```
if (pp < 5000 && config.vestingRecipients.length == 0) {
    revert IssuerVestingRecipientsRequired();
}
```

After that, `_setVestingConfig()` stores vesting recipients and amounts in `PresaleManager`. With `presalePercent == 5000`, `AllocationLib.compute()` returns zero issuer vesting tokens because:

```
presaleTokens = totalSupply * presalePercent / BPS_DENOMINATOR;
liquidityTokens = presaleTokens;
uint256 vestingTotal = totalSupply - presaleTokens - liquidityTokens;
```

```
lpIncentiveTokens = vestingTotal * LP_INCENTIVE_PERCENT / 100;
issuerVestingTokens = vestingTotal - lpIncentiveTokens;
```

At 50%, `presaleTokens + liquidityTokens == totalSupply`, so `vestingTotal == 0` and `issuerVestingTokens == 0`.

When the launch graduates, `PresaleManager.seedLiquidity()` only mints to and initializes the vesting stream if the issuer vesting bucket is non-zero:

```
if (issuerVestingTokens > 0 && vestingStream != address(0)) {
    IBoardwalkToken(token).mint(vestingStream, issuerVestingTokens);
}

if (vestingStream != address(0) && issuerVestingTokens > 0 &&
↪   _vestingRecipients.length > 0) {
    IVestingStream(vestingStream).initialize(token, liquiditySeedTime,
    ↪   _vestingRecipients, _vestingAmounts);
}
```

As a result, an Advanced launch with `presalePercent == 5000` and non-empty vesting recipients records a `vestingStream` address in `LaunchInfo`, but that stream is never initialized and never funded.

Impact

This does not directly steal or lock user funds, because the 50% presale configuration has no issuer vesting allocation to distribute. The impact is inaccurate launch metadata and a permanently unusable vesting contract that may mislead issuers, recipients, indexers, and frontends into treating the launch as if it has a valid vesting stream.

This also breaks the documented clone-count invariant: Advanced launches should deploy 5 clones only when `presalePercent < 50%`, otherwise 4.

Severity: Low.

Code Snippet

- `boardwalk-contracts/src/core/LaunchFactory.sol:290-293`
- `boardwalk-contracts/src/core/LaunchFactory.sol:371-372`
- `boardwalk-contracts/src/core/LaunchFactory.sol:474-491`
- `boardwalk-contracts/src/core/PresaleManager.sol:195-203`
- `boardwalk-contracts/src/core/PresaleManager.sol:221-223`
- `boardwalk-contracts/src/base/AllocationLib.sol:19-23`
- `boardwalk-contracts/SPEC.md:16`

- `boardwalk-contracts/SPEC.md:22`
- `boardwalk-contracts/SPEC.md:242-249`

Tool Used

Manual Review

Recommendation

Reject vesting recipients for Advanced launches when `presalePercent == 5000`, since there is no issuer vesting bucket in that configuration.

For example, add a validation branch in `LaunchFactory._validateConfig()`:

```
if (pp == 5000 && config.vestingRecipients.length > 0) {
    revert VestingNotAllowedAtFullPresaleAllocation();
}
```

Issue L-13: `swapToRaiseToken()` can forward `RAISE_TOKEN` revenue without clearing its accumulated fee accounting

Source: https://github.com/sherlock-audit/2026-04-bmx-apr-24th-2026/issues/36

Summary

`BoardwalkFeeCollector.receiveFees()` records received fees in `accumulatedFees[token]` for any token, including `RAISE_TOKEN`. However, `swapToRaiseToken()` skips `RAISE_TOKEN` entries in the input token list, then forwards the collector's entire `RAISE_TOKEN` balance at the end of the function.

Unlike `forwardRevenue()`, this path does not clear `accumulatedFees[RAISE_TOKEN]`. As a result, the raise token can be forwarded to the treasury and governance vault while its accounting entry remains stale and non-zero.

Vulnerability Detail

Fees received by the collector are recorded by token:

```
function receiveFees(address token, uint256 amount) external {
    IERC20(token).safeTransferFrom(msg.sender, address(this), amount);
    accumulatedFees[token] += amount;
    emit FeesReceived(token, amount);
}
```

This function does not exclude `RAISE_TOKEN`, so `accumulatedFees[RAISE_TOKEN]` can become non-zero when raise-token revenue is registered through `receiveFees()`.

During `swapToRaiseToken()`, the function skips any token entry that equals `RAISE_TOKEN`:

```
if (token == RAISE_TOKEN) {
    unchecked {
        ++i;
    }
    continue;
}
```

For non-raise tokens, successful swaps clear the corresponding accounting entry:

```
accumulatedFees[token] = 0;
emit FeesSwapped(token, balance, amounts[1]);
```

After processing the list, `swapToRaiseToken()` forwards the entire current `RAISE_TOKEN` balance:

```
uint256 toForward = IERC20(RAISE_TOKEN).balanceOf(address(this));
if (toForward > 0) {
    _forwardToTreasuryAndGovernance(toForward);
}
```

This sweep includes any raise-token balance that was previously registered in `accumulatedFees[RAISE_TOKEN]`, but the function does not clear that mapping entry.

The dedicated raise-token forwarding function handles the same accounting correctly:

```
function forwardRevenue() external {
    if (msg.sender != keeper) revert NotKeeper();
    uint256 amount = IERC20(RAISE_TOKEN).balanceOf(address(this));
    if (amount == 0) return;
    accumulatedFees[RAISE_TOKEN] = 0;
    _forwardToTreasuryAndGovernance(amount);
}
```

Therefore, two keeper flows that both forward raise-token revenue leave different accounting states.

Impact

This does not directly cause loss of funds because forwarding uses the actual ERC20 balance, not the `accumulatedFees` mapping. The stale value can still mislead off-chain accounting, keeper dashboards, indexers, or monitoring systems that rely on `accumulatedFees[RAISE_TOKEN]` to determine unforwarded revenue.

The stale value may also persist indefinitely: after `swapToRaiseToken()` drains the `RAISE_TOKEN` balance, a later `forwardRevenue()` call returns early when the balance is zero and does not clear the mapping.

Severity: Low.

Code Snippet

- `boardwalk-contracts/src/core/BoardwalkFeeCollector.sol:71-77`
- `boardwalk-contracts/src/core/BoardwalkFeeCollector.sol:97-101`
- `boardwalk-contracts/src/core/BoardwalkFeeCollector.sol:119-125`
- `boardwalk-contracts/src/core/BoardwalkFeeCollector.sol:134-138`
- `boardwalk-contracts/src/core/BoardwalkFeeCollector.sol:143-148`

Tool Used

Manual Review

Recommendation

Clear `accumulatedFees[RAISE_TOKEN]` whenever `swapToRaiseToken()` forwards the collector's raise-token balance:

```
uint256 toForward = IERC20(RAISE_TOKEN).balanceOf(address(this));
if (toForward > 0) {
    accumulatedFees[RAISE_TOKEN] = 0;
    _forwardToTreasuryAndGovernance(toForward);
}
```

Issue L-14: Integrator claim rate limit becomes ineffective for recurring fee accruals

Source: https://github.com/sherlock-audit/2026-04-bmx-apr-24th-2026/issues/37

Summary

`IntegratorFeeCollector._claimableBySlot()` calculates the 24-hour claim cap as `totalAccrued / 4`. Since `totalAccrued` is a lifetime cumulative value that only increases as more fees arrive, the cap grows with historical accruals rather than with the currently unclaimed balance or each new fee deposit.

As a result, after a slot has accumulated enough historical fees, later recurring accruals can effectively be claimed in full every day. This weakens the intended 25%/24h throttling of integrator fee claims.

Vulnerability Detail

The integrator claim limit is calculated as follows:

```
uint256 unclaimed = state.totalAccrued - state.totalClaimed;
if (unclaimed == 0) return 0;

uint256 maxClaimable = state.totalAccrued / CLAIM_RATE_DIVISOR;
```

For a fresh 24-hour window, the function returns:

```
return _min(maxClaimable, unclaimed);
```

For the same 24-hour window, it compares `claimedInCurrentPeriod` against the same lifetime-based `maxClaimable`:

```
if (state.claimedInCurrentPeriod >= maxClaimable) return 0;
uint256 remainingInPeriod = maxClaimable - state.claimedInCurrentPeriod;
return _min(remainingInPeriod, unclaimed);
```

This means the cap is based on lifetime `totalAccrued`, not the amount that remains unclaimed and not the amount newly accrued during the current period. Once enough historical fees have accrued, the cap can exceed the daily incoming fee amount.

For example, assume a slot receives `200` fees per day and claims as much as allowed each day:

```
Day 1: totalAccrued = 200,  maxClaimable = 50,  unclaimed before claim = 200  ->
 ↪  claim 50
Day 2: totalAccrued = 400,  maxClaimable = 100, unclaimed before claim = 350  ->
 ↪  claim 100
```

```
Day 3: totalAccrued = 600,  maxClaimable = 150, unclaimed before claim = 450  ->
 ↪  claim 150
Day 4: totalAccrued = 800,  maxClaimable = 200, unclaimed before claim = 500  ->
 ↪  claim 200
```

From day 4 onward, the daily cap is at least equal to the new daily accrual of `200`. The cap may still drain the accumulated backlog for a few more days, but it no longer meaningfully limits newly accrued daily fees. Once the backlog is cleared, each later `200` daily accrual can be claimed in full because `totalAccrued / 4` remains greater than the new unclaimed amount.

The same issue can also occur within an existing 24-hour window. If a slot claims its full window allowance and additional fees accrue before the window ends, `totalAccrued` increases immediately, raising `maxClaimable` and allowing additional claims in the same period.

Therefore, the implemented invariant is effectively:

```
claim up to 25% of lifetime totalAccrued per 24h
```

It is not:

```
claim up to 25% of currently unclaimed fees per 24h
```

or:

```
release each newly accrued fee amount over four 24h periods
```

Impact

The issue does not allow an integrator to claim fees belonging to another slot. However, it weakens the intended withdrawal throttling for integrator fees. After a short warm-up period under steady accruals, integrators can claim and swap newly accrued fees in full each day, defeating the expected 25%/24h pacing for future fee inflows.

This can increase token sell pressure earlier than intended and make the rate limit ineffective as an ongoing protection mechanism for launch-token fee claims.

Severity: Low.

Code Snippet

- `boardwalk-contracts/src/core/IntegratorFeeCollector.sol:25-27`
- `boardwalk-contracts/src/core/IntegratorFeeCollector.sol:395-417`
- `boardwalk-contracts/src/core/IntegratorFeeCollector.sol:473-480`
- `boardwalk-contracts/SPEC.md:136-142`

Tool Used

Manual Review

Recommendation

Clarify the intended invariant for integrator claim throttling.

If the intended behavior is to throttle the currently unclaimed balance, calculate the cap from `unclaimed` instead of lifetime `totalAccrued`, with an equivalent dust escape:

```
uint256 maxClaimable = unclaimed / CLAIM_RATE_DIVISOR;
if (maxClaimable == 0) return unclaimed;
```

If the intended behavior is to release each newly accrued amount over four days, track accruals by deposit period or use an accounting model that separates newly accrued fees from historical accrued fees, rather than using lifetime `totalAccrued` as the denominator.

Disclaimers

Sherlock does not provide guarantees nor warranties relating to the security of the project.

Usage of all smart contract software is at the respective users' sole risk and is the users' responsibility.