

Code Assessment of the MetaDEX Relay Smart Contracts

October 21, 2026

Produced for

DROMOS LABS



Contents

1	Executive Summary	3
2	Assessment Overview	4
3	System Overview	6
4	Trust Model	16
5	System Considerations	23
6	Terminology	26
7	Findings	27
8	Limitations and use of report	44

1 Executive Summary

Dear Dromos Labs Team,

Thank you for trusting ChainSecurity with the review of the MetaDEX Relay. As a globally trusted security partner for tier-one DeFi applications, ChainSecurity provides specialized, independent analysis for complex smart contract systems. We greatly appreciate the opportunity to collaborate with Dromos Labs, the core contributing development company behind Aerodrome, the leading decentralized exchange on Base. MetaDEX Relay evolves the Aerodrome design and brings it to a broader set of chains, including Ethereum mainnet and Arbitrum-style networks.

Dromos Labs implements the Relay, a vault that pools users' staked weight in one sAERO and votes it across chains and gauges. Each deposit is split into a principal token claiming the stake and a yield token claiming its rewards. We have conducted a thorough review of the scope with a focus on liveness of withdrawals, share accounting and reward accounting:

User withdrawals can only be processed once enough voting weight has returned from other chains. A leaf operator could delay the return of that weight by repeatedly resetting the gauge allocation cooldown on its chain, see [Leaf Operator Can Block the Return of a Relay's Voting Weight](#). The specification was partially changed and Dromos Labs plans not to use a leaf operator with local voting in its intended setups.

Share accounting is critical because the Relay pools all deposits, so each user must keep their fair share of the pool. The inflation protection was insufficient, since the bootstrap owner could withdraw the seed and inflate the share price so that pending depositors received zero shares, which has been resolved, see [Insufficient Protection Against Inflation Attacks](#).

Rewards arrive in arbitrary tokens on several chains and pass through entrypoints and Relay modules before holders can claim them. Tokens left behind on an entrypoint by a swap batch could be swept by anyone, which has been resolved, see [Intermediate Tokens from a Swap Batch Are Left Behind on the Entrypoint](#).

We provide this audit report to assist in forming an opinion on the risk profile of MetaDEX Relay. In summary, we find that the codebase provides a good level of security, if operated under the assumptions documented in [System Overview](#), [Trust Model](#) and [System Considerations](#).

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 MetaDEX Relay 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 September 2026	17dec51b133bc5e4dc6317cfe891563efe998c02	Initial review
2	30 September 2026	9ee7bac2fb25f2cdbc2cbd9807f3c62706fb3e1e	Second version
3	15 October 2026	c70736bfc8c06a5dddf692bb2117fa8aea1a635	Fix Review
4	19 October 2026	79c8d5b62b7558c7fd58261523e517c588c0378d	Final Version

For the Solidity smart contracts, the compiler version `0.8.36` was chosen.

```
V3/src/
├── relay/
│   ├── entrypoints/
│   │   ├── BaseEntrypoint.sol
│   │   ├── Compounder.sol
│   │   ├── MultiConverter.sol
│   │   ├── MultiEntrypoint.sol
│   │   ├── MultiHybrid.sol
│   │   ├── SingleConverter.sol
│   │   └── SingleHybrid.sol
│   ├── leaf/
│   │   ├── RelayLeafModule.sol
│   │   └── RelayLeafModuleDeployer.sol
│   ├── libraries/
│   │   ├── AllocationLib.sol
│   │   ├── DenseQueue.sol
│   │   ├── QueueLib.sol
│   │   ├── RelayConfigLib.sol
│   │   ├── RelayGovernanceLib.sol
│   │   ├── RelayInitLib.sol
│   │   ├── RelayModuleLib.sol
│   │   └── RelayRewardsLib.sol
│   ├── MaxiRelay.sol
│   ├── ProtocolRelay.sol
│   ├── RelayBase.sol
│   ├── RelayFactory.sol
│   ├── RelayIcaModule.sol
│   ├── RelayIcaModuleDeployer.sol
│   ├── RelayRoles.sol
│   ├── RelayToken.sol
│   ├── RelayTokenVotes.sol
│   └── RelayVoteAdapter.sol
```

In **Version 2** the following files have been added to the scope:

```
V3/src/  
└─ relay/  
    └─ libraries/  
        └─ RelayRakeLib.sol
```

In **Version 3** the following files have been added to the scope:

```
V3/src/  
└─ relay/  
    └─ RelayFactoryRegistry.sol
```

In **Version 4** the following files have been added to the scope:

```
V3/src/  
└─ relay/  
    └─ tape/  
        └─ RelayTape.sol
```

2.1.1 Excluded from scope

All contracts not explicitly listed above are out of scope of this review.

Third-party libraries, namely OpenZeppelin, Solady and Nomad, are out of scope and assumed to work correctly. Further, while the integration with Hyperlane is in scope of this review, the Hyperlane contracts themselves are not. Their security depends on a correct configuration, which is also not in scope of this review.

The economic design and the correct choice of parameters are also out of scope of this review.

The contracts are expected to be deployed on Arbitrum, Robinhood Chain, Ethereum Mainnet, Optimism, Arc, Ink and Base. Base is the root chain, where the Relays are deployed. All other chains are leaf chains, which host the `RelayLeafModule` and the interchain accounts of `RelayIcaModule`. Other chains are not covered by this review, and the differences in their behavior must be carefully considered before deploying there.

3 System Overview

This system overview describes the latest received version (**Version 2**) of the contracts as defined in the [Assessment Overview](#).

At the end of this report section, we have added subsections for each of the changes according to the versions.

Furthermore, in the findings section, we have added a version icon to each of the findings to increase the readability of the report.

Dromos Labs offers MetaDEX Relay, a vault (called Relay) that pools depositors' staked weight in its own sAERO staking NFT. Depositors receive a principal token (PT) as a claim on the weight and a yield token (YT) claiming distributed rewards (PT / YT). `VOTER_ROLE` allocates the weight to chains and gauges. The votes earn the Relay fees and incentives. Fees are a share of the voted pools' trading fees. Incentives are tokens third parties deposit to attract votes. Entrypoint contracts compound these rewards into the sAERO, raising every PT's value, or distribute them to YT holders (Entrypoints). A rake on both paths pays the protocol and the allocator (Rake). The sAERO can still be used to vote on governance proposals (Governance).

`RelayFactory` creates `MaxiRelay` and two-level `ProtocolRelay` Relays ([RelayFactory](#), [Relay Tiers](#)). A Relay is permanent or timed, like its sAERO's lock ([Lock Refresh](#)). The Relay owner and roles gate privileged calls ([Access Control and Roles](#)).

3.1 Integrated Aerodrome Contracts

The Relay builds on the following Aerodrome contracts, which are not in scope ([Assessment Overview](#)). The root chain hosts the `VotingEscrow`, the `Voter` and every Relay contract. Leaf chains are reached by cross-chain messages.

- `VotingEscrow`: issues sAERO NFTs, which lock `TOKEN`, the protocol's ERC-20 token, permanently or until an end time. Weight denotes the staked `TOKEN` amount, not its voting power.
- `Voter`: records each sAERO's weight per chain. Chain 0 is its entry for an sAERO's unallocated weight. [Allocation](#) moves weight from chain 0 to a chain, root or leaf.
- `LeafVoter`: splits the weight allocated to its chain across gauges by gauge votes.
- `VoterPaymentsModule` (VPM): moves chain 0 weight between sAEROs for a fee.

3.2 PT / YT

Only the Relay mints and burns PT and YT. The price of one share (one PT) is

$$p = \frac{B}{S}$$

where B is `totalBacking` (accounted weight of the sAERO) and S is the PT supply. Deposits and withdrawals (exits) wait in queues. Both are priced when processed.

At creation, B equals the creator's seed stake, at least `MIN_STAKE_AMOUNT` (`1e18`). `bootstrapOwner` receives as many PT, but no YT ([RelayFactory](#)). So p starts at 1. As an exit from an open Relay needs as many YT as PT ([Withdrawals](#)), the seed shares can only leave once the Relay is closed, unless `bootstrapOwner` obtains YT, which is only possible where YT is transferable.

A deposit of net weight a mints a/p PT and as many YT ([Deposits](#)). An exit of s shares burns s PT and YT and pays $s \cdot p$, less the VPM fee ([Withdrawals](#)). Both round down, so rounding never lowers p . After closure only PT is minted and burned ([Closure and Evacuation](#)).

PT is never transferable and holds the Relay's governance votes ([Governance](#)). YT entitles its holder to distributed rewards ([Distribution](#)). It is transferable only on a `MaxiRelay` created with `ytTransferable` set.

3.3 Deposits

A deposit takes two steps.

1. `requestDeposit(tokenId, amount[, recipient])` is permissionless. It moves `amount` from the source's chain 0 into the Relay's sAERO at once, less the VPM fee. The net is queued as a pending deposit. It reverts on a closed Relay, or if the net is below `minDeposit` or worth zero shares at the current p .
2. `processPending(maxEntries)` is restricted to `KEEPER`. It admits entries in request order, each at the current p . It mints PT and YT (only PT on a closed Relay) and adds the net to `totalBacking`. If p rose after the request, an entry can round to zero shares. It is still admitted, and its weight goes to existing holders.

Anyone can admit entries at least `keeperWindow` old with `processOverduePending(ids)`. A request cannot be canceled.

The caller, the Relay and its current VPM ([Configuration](#)) must each own or be approved for the source sAERO. The `VotingEscrow` refuses to move weight into a shorter lock, so a timed Relay accepts only timed sources that end no later than it does. The shares go to `recipient`, by default the source's owner at request time. `ProtocolRelay` requires an allow-listed recipient ([Relay Tiers](#)).

Until admission, queued depositors receive no PT or YT and have no share in the rewards earned. Their weight sits on the Relay's chain 0, where `allocate()` (`VOTER_ROLE`) and exits can use it ([Allocation](#), [Withdrawals](#)).

3.4 Withdrawals

A withdrawal takes two steps.

1. `registerOnWithdrawQueue(shares, destination)` is permissionless. It escrows the shares and appends the exit to the queue. The holder needs that many unescrowed PT and, while the Relay is open, as many unescrowed YT. `shares` must be non-zero. The holder can always exit all unescrowed PT. A partial exit must be at least `minWithdrawal` and leave at least `minWithdrawal` unescrowed PT behind.
2. `processWithdrawals(maxEntries)` is permissionless. It pays exits in queue order at the current p ([PT / YT](#)). It sends the weight, less the VPM fee, to the destination and burns PT and YT (PT only once the Relay is closed).

An exit cannot be canceled. Escrowed PT still votes and escrowed YT still earns rewards.

If the destination is `MINT_SENTINEL` or lacks a lock at least as long as the Relay's ([Deposits](#)), the holder receives the payout as a fresh sAERO with the Relay's lock.

Only the Relay's chain 0 weight ([Allocation](#)) can be used to process exits. `processWithdrawals()` stops at the first exit that chain 0 cannot fully pay ([Closure and Evacuation](#)).

3.5 Allocation

`allocate(chainDispatches, gaugeDispatches, refundRecipient)` is restricted to `VOTER_ROLE`. Chain dispatches move chain 0 weight to other chains. The Relay caps the weight available to allocate at the chain 0

weight minus a reserve for queued exits. The reserve is the queued exits' pro-rata share of the sAERO's whole stake, which also includes queued deposits and weight sent directly to the sAERO. Gauge dispatches overwrite the Relay's gauge votes on a chain. The `Voter` and `LeafVoter` validate all dispatches, including an allocation cooldown between an sAERO's gauge votes on a chain.

On an open Relay, `VOTER_ROLE` can return weight to chain 0 by voting it to the sentinel gauge `DEALLOC_GAUGE`.

The Relay owner can set one operator per chain, root included ([Configuration](#)). The operator can claim the chain's rewards ([Inflow](#)). Where the chain enables local voting, the operator can re-vote the Relay's weight there under the same cooldown, and return it with `DEALLOC_GAUGE`.

3.5.1 Lock Refresh

The permissionless `extendLock()` refreshes a timed Relay's lock, even after closure. `allocate()` and `compound()` (`COMPOUNDER`) run the refresh before allocating or compounding. At most once a week, it moves the lock end to this week's start plus `lockWeeks` weeks. On a permanent Relay, `lockWeeks` is 0 and the refresh has no effect.

3.6 Closure and Evacuation

Closure is the depositors' escape hatch when exits are not processed. `close()` is one-way. The Relay owner can call it at any time. Anyone else can call it only while the oldest withdrawal request is at least `evacuationWindow` old and its payout at the current p exceeds the Relay's chain 0 weight ([Withdrawals](#)).

Once closed:

- `requestDeposit()` and `allocate()` revert.
- Queued deposits are still admitted, but mint PT only. Exits burn PT only, so YT can keep claiming rewards. The seed shares can exit ([PT / YT](#)).
- `compound()`, `notifyReward()`, `claim()`, `claimRewards()`, the fee collections ([Rake](#)), `expressVote()` and the [entrypoints](#) keep working ([Distribution](#)).

The permissionless `evacuate()` works only on a closed Relay. It votes the Relay's full allocated weight on one chain to `DEALLOC_GAUGE`. Chain 0 is credited on the leaf's reply, or at once on the root chain. The chain refuses the vote during its allocation cooldown ([Allocation](#)).

`emergencyDeallocate()` is restricted to `KEEPER` on an open Relay and permissionless on a closed one. It works only on a chain the `Voter`'s `CHAIN_STATUS_ROLE` has suspended with emergency deallocation allowed. It credits the chain's allocated weight to chain 0 in the same call.

After `close()`, anyone can call `evacuate()` on each chain, then `processWithdrawals()`. `evacuate()` reverts on a suspended chain, which needs `emergencyDeallocate()` instead. The escape is bounded by `evacuationWindow` and each chain's allocation cooldown, as long as no operator's local re-vote restarts that chain's cooldown. The [Trust Model](#) states the other conditions it relies on.

3.7 Rewards

Fees and incentives are claimed ([Inflow](#)), then compounded or distributed ([Distribution](#)), less the rake ([Rake](#)).

3.7.1 Inflow

The permissionless `claimRewards()` claims one chain's rewards through the `Voter`. The caller never chooses the recipient.

1. On root, the Relay receives them in the same transaction.
2. On a leaf chain, the chain's leaf recipient receives them later. The Relay owner sets it ([Configuration](#)). It can be a `RelayLeafModule`, a `RelayIcaModule`'s interchain account or any other address ([Relay Modules](#)). Without one, the claim reverts.

A chain's operator ([Allocation](#)) can also claim that chain's rewards, to any recipient.

3.7.2 Distribution

`compound(amount)` is restricted to `COMPOUNDER`. It runs the [lock refresh](#), then stakes `TOKEN` the Relay holds, less the rake, into its sAERO. `totalBacking` rises without minting, so every PT gains value (PT / YT).

`KEEPER` adds and removes the tokens `notifyReward()` may distribute, with `addRewardToken()` and `removeRewardToken()`. The tier caps their number ([Relay Tiers](#)). Only a token never distributed can be removed, so accounted rewards can never be stranded.

`notifyReward(token, amount)` is restricted to `CONVERTER`. It distributes a registered token the Relay holds, less the rake, to all current YT holders pro rata. A compound or notify credits the holders at the time of the call, not those who held while the rewards accrued.

`claim(token, to)` is permissionless. It pays the caller's whole pending `token` reward to any address.

`accountedBalance[token]` tracks notified tokens not yet claimed and rake not yet collected. `pull()`, `compound()`, `sweep()` and `notifyReward()` use only the unaccounted rest of the balance, so owed tokens leave only through `claim()` and the fee collections ([Rake](#)). Besides these and `compound()`, two functions move tokens out:

1. `pull(token, amount)` is restricted to `COMPOUNDER` or `CONVERTER`. It sends any token to the caller, with no return enforced ([Entrypoints](#)).
2. `sweep(legs)` is restricted to `SWEEPER` and works only on a level 2 `ProtocolRelay`. It sends any ERC-20 to recipients the caller chooses.

3.7.3 Rake

`compound()` and `notifyReward()` take two rakes from `amount` before the rest reaches the holders. The protocol rake is `protocolPips` of `amount`. The allocator rake is `allocatorPips` of the remainder. Both round down. Each rate is capped by `MAX_RAKE_PIPS`, an immutable below 100%. The rake stays in the Relay, booked per token in `feeAccrued` and counted in `accountedBalance` ([Distribution](#)), until collected:

1. `collectProtocolFees(tokens)` is permissionless. It sends the protocol part to the `rakeTreasury` of the Relay's `factory()` ([RelayFactory](#)).
2. `collectAllocatorFees(tokens, recipient)` is restricted to `VOTER_ROLE`. It sends the allocator part to any recipient.

The creator sets the initial rates and `RAKE_MANAGER_ADMIN`. The admin seats and removes `RAKE_MANAGER` holders with `setRakeManager()` and hands over its seat with `transferRakeManagerAdmin()`. A

`RAKE_MANAGER` proposes new rates with `proposeRakes()`. A pair that raises neither rate applies at once and drops any pending pair. Any other pair waits `entrypointTimelock`, then a `RAKE_MANAGER` applies it with `executeRakes()`. A new proposal replaces the pending one and restarts the delay.

3.8 Entrypoints

Entrypoints convert tokens held by the Relay into backing through `compound()` or into YT rewards through `notifyReward()` ([Distribution](#)). `Compounder` holds `COMPOUNDER`. `SingleConverter` and `MultiConverter` hold `CONVERTER`. `SingleHybrid` and `MultiHybrid` hold both.

Each endpoint is bound at deployment to one Relay, `RELAY()`, and acts only on it. `RelayFactory` and `proposeEntrypoint()` refuse an endpoint bound to another Relay ([RelayFactory](#), [Configuration](#)). The swap, idle and sweep functions can only be called by the bound Relay's `KEEPER`.

A swap (`swapAndCompound()`, `swapAndConvert()`) pulls `tokenIn` from the Relay and then runs `KEEPER`'s plan on a `KEEPER`-chosen Metarouter the `FactoryRegistry` approves. The output must reach the non-zero `minAmountOut` that `KEEPER` provides. The endpoint sends the output and all its remaining `tokenIn` to the Relay. `swapAndCompound()` then calls `compound()` for the output. `swapAndConvert()` calls `notifyReward()` for the Relay's whole unaccounted balance of the target token, which includes the output. The idle paths (`compoundIdleBalance()`, `convertIdleBalance()`) skip the swap. They pass the Relay's whole unaccounted balance of the output token, such as [bridged leaf proceeds](#), to `compound()` or `notifyReward()`. `sweepToRelay(tokens)` sends the endpoint's whole balance of each token to the Relay, such as intermediate tokens a swap route left behind.

- Single (`Compounder`, `SingleConverter`, `SingleHybrid`) endpoints: have no mutable configuration. `SingleConverter` and `SingleHybrid` have an immutable target token.
- Multi (`MultiConverter`, `MultiHybrid`) endpoints: are meant for a level 2 `ProtocolRelay` ([Relay Tiers](#)). The Relay owner can change their target tokens and set input tokens they refuse to sell.

The Hybrids store a compound weight, the share of rewards `KEEPER` should compound rather than distribute. It is informational only. No function reads it and nothing enforces it on-chain. It is immutable on `SingleHybrid`. On `MultiHybrid` the Relay owner can change it with `setCompoundWeight()`.

3.9 Relay Modules

Relay modules sell a Relay's [leaf-chain rewards](#) into one token per chain and bridge it to the Relay on root, where an [endpoint](#) compounds or distributes it. Anyone can deploy a module.

In both modules, the keeper names the bridge's `targetRouter` on root. Zero keeps the bridge's default route. A non-zero target can deliver a different token. Hence, the keeper chooses which token the Relay receives.

3.9.1 RelayLeafModule

`RelayLeafModule` runs on a leaf chain. `RelayLeafModuleDeployer` fixes each module's `OUT_TOKEN` and Hyperlane warp route (token bridge). The module owner grants and revokes the module's own `KEEPER` role.

The module receives rewards as leaf recipient ([Inflow](#)), or through its permissionless `claim()` when set as operator ([Allocation](#)).

The module `KEEPER` sells tokens into `OUT_TOKEN` with `swap()`. The module `KEEPER`'s `bridge(metarouter, maxFee, targetRouter)` sends the whole `OUT_TOKEN` balance to the Relay on root through `targetRouter` and caps the bridge's fee at `maxFee`. If `OUT_TOKEN` is the Metarouter's native ERC-20, such as USDC on Arc, the attached message fee is not bridged. Swaps and bridges need a Metarouter the `FactoryRegistry` approves.

The module `KEEPER`'s `redeem(gasLimit)` burns the module's whole `RECEIPT_TOKEN` balance, the `LeafVoter`'s receipt token. Root then pays as much `TOKEN` to the Relay.

The module owner's `rescueOutToken()` sends all `OUT_TOKEN` to any address without delay. The module `KEEPER`'s `sweepNative()` and `sweepToken()` send any other balance, except `RECEIPT_TOKEN`, to any

address. `sweepNative()` reverts if it lowers the `OUT_TOKEN` balance, as when the native asset is also an ERC-20.

3.9.2 RelayIcaModule

`RelayIcaModule` runs on root and controls a Hyperlane interchain account (ICA) on each leaf chain. The Relay owner sets that ICA, not the module, as the chain's leaf recipient ([Inflow](#)).

The Relay owner sets the `keeper` and each leaf chain's `LeafConfig` of Metarouter, output token and bridge. The leaf Metarouter needs no `FactoryRegistry` approval. The `keeper` is separate from the Relay's `KEEPER`.

With `dispatchPlan()`, the `keeper` alone submits swaps for a leaf chain. The module builds calls that bridge the whole output to the Relay on root through the plan's `targetRouter` and cap the warp route's fee at one percent. The `keeper` does not choose the destination, which is always the Relay. The last calls revoke the Metarouter's approval for each token sold. The module emits the calls and sends only their hash to the leaf. Anyone can reveal and run them until the plan's `leafDeadline`, even after the `keeper` or `LeafConfig` is cleared. The `keeper`'s `sweepNative()` and `sweepToken()` send the module's root balances to any address.

In the figure below, the plan sells `tokenA` twice under one approval, in a concentrated-liquidity (CL) pool and in a V2 pool. It sells `tokenB` in a CL pool. The plan opens with the approvals and closes with an `approve(metarouter, 0)` call for each of them, which resets the allowances. The dashed calls are optional and only built when `sweepHeldOutToken` is true. They add the output token the ICA already holds to the bridged amount, and approve and reset the Metarouter's allowance for that token.

RemoteCall[] _calls (RelayIcaModule._buildPlan)

#	to.data	value
[0]	<code>tokenA.approve(metarouter, a1 + a2)</code>	0
[1]	<code>tokenB.approve(metarouter, b1)</code>	0
[2]	<code>outToken.approve(metarouter, type(uint256).max)</code>	0
[3]	<code>metarouter.execute(commands, inputs, leafDeadline)</code> [0] CL_SWAP_EXACT_IN tokenA, Amount a1, payerIsUser [1] V2_SWAP_EXACT_IN tokenA, Amount a2, payerIsUser [2] CL_SWAP_EXACT_IN tokenB, Amount b1, payerIsUser [3] FUND_ERC20 outToken, 100% [4] BRIDGE_TOKEN outToken, 100% -> Relay on root	leafMessageFee
[4]	<code>outToken.approve(metarouter, 0)</code>	0
[5]	<code>tokenA.approve(metarouter, 0)</code>	0
[6]	<code>tokenB.approve(metarouter, 0)</code>	0

 always built  only if `sweepHeldOutToken`

3.10 Governance

The permissionless `expressVote()` implements Flexible Voting for the protocol's `Governor`. Each caller casts its slice of the Relay sAERO's voting power. The slice is the total power times the caller's delegated [PT](#) votes over the PT supply, all at the proposal snapshot the vote adapter reports. PT holders are self-delegated by default. The power includes queued deposit weight, which existing PT holders can cast.

A caller either puts its whole remaining slice on one option or splits it across For, Against and Abstain. The Relay always forwards the vote as a split, which the adapter encodes as the `Governor` call. The Relay sends it

only if its selector is `castVoteWithReasonAndParams`. `setGovernor()` (Relay owner) replaces the `Governor` and adapter with no delay.

3.11 Relay Tiers

The three Relay types are `MaxiRelay` and level 1 and level 2 `ProtocolRelay`. Level 1 is meant to hold external depositors' funds, so it refuses `sweep()`.

Feature	<code>MaxiRelay</code>	Level 1	Level 2
Deposit recipients	Anyone	Allow-listed	Allow-listed
YT transfers	Only with <code>ytTransferable</code>	Never	Never
Reward tokens	At most 1	At most 1	At most 10
<code>COMPOUNDER</code> / <code>CONVERTER</code>	Fixed at creation, at least one	Fixed at creation, at least one	Optional at creation, then timelocked attachment and immediate revocation
<code>sweep()</code> (SWEEPER)	No	No	Yes

On `ProtocolRelay`, only the Relay owner can call `setAllowList()`, `kick()` and `promoteToLevel2()`. The allow-list gates only the recipient of `requestDeposit()`. `kick()` delists an account and queues its unescrowed shares for exit to a fresh sAERO ([Withdrawals](#)).

`promoteToLevel2()` is one-way and immediate. It transfers ownership to any non-zero address. Role holders and shares stay unchanged. A `ProtocolRelay` can also start at level 2 ([RelayFactory](#)).

3.12 RelayFactory

Holders of `RELAY_DEPLOYER_ROLE` on the `Voter` call `createMaxiRelay()` or `createProtocolRelay()`. Only the latter can start the Relay at level 2 ([Relay Tiers](#)). In one transaction, the factory stakes the caller's seed `TOKEN` into a new sAERO, then deploys and initializes a clone of the tier contract. It sets `lockWeeks` to the seed's lock length in weeks, 0 if permanent. The Relay starts on the factory's VPM, `Governor` and vote adapter. The caller names the Relay owner, the initial role holders, the initial rake rates ([Rake](#)) and `bootstrapOwner` ([PT / YT](#)).

The Relay's address is known before creation through `predictRelay()`. A named `COMPOUNDER` or `CONVERTER` must already be deployed and bound to that address ([Entrypoints](#)). The factory records them, readable through `initialEntrypoints()`.

The factory holds `rakeTreasury`, which receives the protocol rake of every Relay it initialized. `setRakeTreasury()` is restricted to `GOVERNANCE_ROLE` on the `Voter` and changes it with no delay.

`GOVERNANCE_ROLE` also sets a relay tape and its gas limit with `setRelayTape()`, with no delay. While a tape is set, `compound()` and `notifyReward()` of every Relay the factory created report to it through the factory. They revert if less gas than the gas limit is left, but a failing tape does not block them ([RelayTape](#)).

`minDeposit` (at least `MIN_STAKE_AMOUNT`), `minWithdrawal`, `keeperWindow`, `evacuationWindow`, `entrypointTimelock` (each non-zero), `lockWeeks`, `MAX_RAKE_PIPS`, the PT, the YT, YT transferability and the tier contract have no setter.

`initialize()` is permissionless, so Relays can also be created outside the factory. Its caller becomes the Relay's `factory()`, which names the rake treasury. Only factory Relays pass `isRelay()`.

3.12.1 RelayFactoryRegistry

`RelayFactoryRegistry` holds the set of approved Relay factories. `GOVERNANCE_ROLE` on the `Voter` adds and removes factories with `registerRelayFactory()` and `unregisterRelayFactory()`, with no delay. The Metarouter accepts a deposit into a Relay only if the Relay's `factory()` is approved and that factory's `isRelay()` confirms the Relay. Unregistering a factory only stops Metarouter deposits into its Relays. It does not affect `RelayFactory` creation or `requestDeposit()` called on the Relay itself.

3.12.2 RelayTape

`RelayTape` records what each Relay pays its holders, as data for computing a Relay's APR. Only factories on its allowlist can write to it, and `RelayFactory` only forwards reports from Relays it created. Each reward token of a Relay has its own reward series, and each Relay has one backing series. `notifyReward()` reports the amount to be distributed after rake and the new reward index. `compound()` reports the amount staked after rake, the new `totalBacking` and the principal supply, from which the tape derives the backing per share, scaled by $1e27$.

Every report adds its amount to the series' running total, which wraps at 128 bits. The tape then stores an observation of the running total, the timestamp and the per-share value, capped at 224 bits. It writes the hourly ring of 48 slots at most once per hour, and the daily ring of 32 slots at most once per UTC day.

`observeReward()` and `observeBacking()` return, for each lookback, the newest stored observation at or before that time. Lookbacks of up to 48 hours read the hourly ring, longer ones the daily ring. If no observation is that old, the oldest one is returned. A lookback of zero returns the running total and the per-share value read live from the Relay, or the newest stored backing observation if the principal supply is zero.

`dailyRewardObservations()` and `dailyBackingObservations()` return the daily ring, oldest first.

The tape owner (`Ownable2Step`) sets the allowlist with `setAllowedFactory()`, with no delay. The deployer seeds the initial allowlist.

3.13 Configuration

The Relay owner's `setName()` renames the Relay, not PT and YT. `KEEPER`'s `setVoterPaymentsModule()` switches the Relay's `vpm` to any VPM the `VotingEscrow` authorizes. Depositors must approve the new VPM on their source sAERO ([Deposits](#)).

Three Relay owner settings are proposed, then executed after `entrypointTimelock`:

1. The leaf recipient per chain, except the root chain ([Inflow](#)).
2. The operator per chain ([Allocation](#)).
3. An entrypoint role on a level 2 `ProtocolRelay`, for an entrypoint bound to the Relay ([Relay Tiers](#), [Entrypoints](#)).

The Relay owner's `cancelLeafRecipient()` and `cancelOperator()` drop a pending proposal and leave the current setting in place. `clearLeafRecipient()` disables the chain's claims with no timelock and leaves a pending proposal in place. `revokeOperator()` and `revokeEntrypoint()` need no timelock and also cancel the pending proposal. `ENTRYPOINT_VETOER` can cancel a pending entrypoint proposal with `vetoEntrypoint()`. The Relay owner sets and clears [RelayIcaModule](#) settings the same way. Setting or revoking an operator takes effect only when the `Voter`'s message arrives. Rake raises wait the same `entrypointTimelock` ([Rake](#)).

3.14 Access Control and Roles

Each Relay has one owner and eight roles: `KEEPER`, `VOTER_ROLE`, `COMPOUNDER`, `CONVERTER`, `SWEEPER`, `ENTRYPOINT_VETOER`, `RAKE_MANAGER` and `RAKE_MANAGER_ADMIN`. `SWEEPER` and `ENTRYPOINT_VETOER` act

only on `ProtocolRelay`. The owner's `grantRoles()` and `revokeRoles()` accept only `KEEPER`, `VOTER_ROLE` and `SWEEPER`. After creation, `COMPOUNDER` and `CONVERTER` change only through level 2 attachment and revocation ([Relay Tiers](#)). `RAKE_MANAGER` changes only through `RAKE_MANAGER_ADMIN`, which in turn moves only by transfer ([Rake](#)). A `RAKE_MANAGER` can renounce its role. `renounceOwnership()` is disabled.

The owner also controls the timelocked [settings](#), the Multi [entrypoint](#) settings, each chain's [operator](#) and the `RelayIcaModule` `keeper`. `RELAY_DEPLOYER_ROLE` on the `Voter` gates `RelayFactory`, and `GOVERNANCE_ROLE` on the `Voter` sets its rake treasury. `RelayLeafModule` has its own owner and module `KEEPER`. `RelayTape` has its own owner, who sets the factory allowlist. The [Trust Model](#) lists each role's permissions and trust level.

3.15 Changes in Version 2

In `Version 2`, the following changes were made.

Rewards and fees:

- Rake added: `compound()` and `notifyReward()` take a protocol and an allocator rake, managed by the new `RAKE_MANAGER` and `RAKE_MANAGER_ADMIN` roles and collected to the factory's `rakeTreasury` and by `VOTER_ROLE` ([Rake](#)).

General relay logic:

- Donations removed: in `Version 1`, `KEEPER`'s `processDonations()` added weight sent directly to the Relay's `saERO` to `totalBacking`, which raised the value of every PT. `Version 2` removes this functionality, together with the `pendingDepositWeight` counter it relied on. Such weight is no longer added to `totalBacking` and never raises the share price. It still sits on chain 0, where `VOTER_ROLE` can allocate it and exits can use it, and the rewards it earns reach holders through compounding or distribution. `TOKEN` sent to the Relay itself can still be compounded into `totalBacking` ([PT / YT](#)).
- The reward accumulator's scale `ACC_SCALE` rose from `1e18` to `1e27`, and reward settlement uses full-precision multiplication and division.
- `swapAndConvert()` notifies the Relay's whole unaccounted balance of the target token rather than only the swap output ([Entrypoints](#)).
- `kick()` no longer settles the holder's rewards itself. They settle when the exit burns the YT, as for any other exit.

Seed and queues:

- `bootstrapOwner` receives only PT for the seed, no YT, so the seed shares cannot exit an open Relay unless YT is obtained elsewhere ([PT / YT](#)).
- The seed stake and `minDeposit` must each be at least `MIN_STAKE_AMOUNT`, instead of non-zero.
- A partial exit must now also leave at least `minWithdrawal` unescrowed PT behind. Exiting the whole unescrowed position is always allowed ([Withdrawals](#)).
- `processOverduePending()` skips entries already admitted instead of reverting, and reverts only if it admits none ([Deposits](#)).
- The withdrawal reserve in `allocate()` is priced against the `saERO`'s whole stake instead of the stake less queued deposits ([Allocation](#)).

Entrypoints:

- Every entrypoint is bound at deployment to one Relay (`RELAY()`), so the Single variants no longer serve many Relays.
- `RelayFactory` and `proposeEntrypoint()` refuse an entrypoint bound to another Relay.

- `KEEPER`'s new `sweepToRelay()` returns tokens left on an entrypoint to its Relay, and swaps emit a `Swapped` event.

Configuration:

- The new `cancelLeafRecipient()` cancels a pending leaf recipient proposal. `clearLeafRecipient()` no longer cancels it. A leaf recipient can no longer be proposed for the root chain ([Configuration](#)).

Relay modules:

- `RelayIcaModule` plans now revoke the Metarouter's approval for each token sold after the swaps ([RelayIcaModule](#)).
- `RelayLeafModule.sweepNative()` reverts if it lowers the `OUT_TOKEN` balance ([RelayLeafModule](#)).

3.16 Changes in Version 3

In `Version 3`, the following changes were made.

General relay logic:

- The Relay owner can `close()` the Relay at any time instead of `KEEPER` ([Closure and Evacuation](#)).
- `compound()` and `notifyReward()` additionally report to a relay tape that `GOVERNANCE_ROLE` sets on `RelayFactory` ([RelayFactory](#), [RelayTape](#)).

RelayFactoryRegistry:

- `RelayFactoryRegistry` has been added ([RelayFactoryRegistry](#)).

Relay modules:

- The keepers of both modules additionally name the bridge's `targetRouter`, which sets the token the Relay receives ([Relay Modules](#)).

RelayLeafModule:

- The permissionless `bridge(metarouter)` and `MIN_BRIDGE_AMOUNT` are removed. Only the module `KEEPER` can bridge, and it selects the `targetRouter`, which sets the token the Relay receives ([RelayLeafModule](#)).
- `bridge()` no longer bridges the attached message fee when `OUT_TOKEN` is the Metarouter's native ERC-20 ([RelayLeafModule](#)).
- The module `KEEPER` can additionally `redeem()` the module's `RECEIPT_TOKEN` to the Relay on root. `sweepToken()` refuses `RECEIPT_TOKEN` ([RelayLeafModule](#)).

3.17 Changes in Version 4

In `Version 4`, `RelayTape` has been added to the scope ([RelayTape](#)).

4 Trust Model

Dromos Labs stated three guarantees for users of MetaDEX Relay: the Relay cannot steal user principal, user principal cannot get permanently stuck, and accounted rewards cannot be stolen. Rewards that are not yet accounted for (not yet compounded or notified) are not covered by these guarantees. The privileged roles are trusted to convert and account them in a timely manner to ensure rewards are fairly distributed among depositors. The code provides these guarantees as follows:

- 1. User principal cannot be stolen:** principal is staked weight tracked by `totalBacking`, which only decreases when an exit burns the exiting holder's own shares, and shares are only minted against weight that actually arrived. No Relay role can move the staked position or mint shares for itself.
- 2. User principal cannot get permanently stuck, only delayed:** anyone may admit a deposit with `processOverduePending()` after `keeperWindow`, and process exits with `processWithdrawals()` as long as enough `TOKEN` is free on chain 0. If an exit stays unpaid for `evacuationWindow`, anyone may `close()` the Relay and bring all `TOKEN` back for withdrawal with `evacuate()`, or with `emergencyDeallocate()` on suspended chains.
- 3. Accounted rewards cannot be stolen:** compounded rewards become staked `TOKEN` tracked by `totalBacking`, and notified rewards are reserved in `accountedBalance`, together with the uncollected rake. `pull()`, `sweep()`, `compound()` and `notifyReward()` only reach `balanceOf(this) - accountedBalance`, which excludes all of them. `accountedBalance` only decreases when a holder claims or a rake is collected.

These guarantees only hold within the Relay contracts. Users still need to fully trust the Aerodrome contracts the Relay integrates with, such as the `VotingEscrow`, the `Voter`, the `LeafVoter` and the `VoterPaymentsModule`, as well as Hyperlane ([External Dependencies](#)), and by extension their privileged roles ([Roles Defined Outside Scope](#)).

4.1 Roles

4.1.1 Relay and Entrypoint Roles

Relay Owner

- Trust level: Trusted.
- Permissions: Gives out, and thereby inherits the privileges of, every Relay-side role except `ENTRYPOINT_VETOER`, `RAKE_MANAGER` and `RAKE_MANAGER_ADMIN`. Gives out `KEEPER`, `VOTER_ROLE` and `SWEeper` with no delay. Assigns the leaf operator and leaf recipient, the `RelayIcaModule` keeper and leaf configs, and on a level 2 `ProtocolRelay` the entrypoints bound to the Relay, behind `entrypointTimelock`. Also calls `close()` at will, `setGovernor()`, `setName()`, the Multi entrypoint settings, and on `ProtocolRelay` `setAllowList()`, `kick()` and the one-way `promoteToLevel2()`.
- Assumptions: `entrypointTimelock` is long enough for depositors to react.
- Worst-case impact: Holds the worst case of every role it can give out. On top of that, it can force-exit any holder through `kick()` and repoint governance votes through `setGovernor()`, both with no delay. Closes a healthy Relay permanently in one call, which stops deposits and allocations but not exits or governance voting. It can also stop every claim on a leaf chain with no delay: `clearLeafRecipient()` makes `claimRewards()` revert for that chain, and `revokeOperator()` removes the operator's claim path once the `Voter`'s message arrives. If it then proposes a new leaf recipient or operator, all rewards that accrue on that chain during `entrypointTimelock` go to the new address, as nobody can claim them to the current one first. The timelock therefore does not protect unclaimed rewards.

Entrypoints (COMPOUNDER and CONVERTER on RelayBase)

- Trust level: Trusted.
- Permissions: Both roles can pull tokens with `pull()`, moving any token balance above `accountedBalance` to themselves. `compound()` for COMPOUNDER and `notifyReward()` for CONVERTER compound or distribute rewards.
- Assumptions: Roles are given to the entrypoint contracts bound to the Relay.
- Worst-case impact: Calls `pull()` directly and takes the whole unaccounted balance of every ERC-20 the Relay holds.

KEEPER (on RelayBase)

- Trust level: Trusted.
- Permissions: `processPending()`, `addRewardToken()`, `removeRewardToken()` and `setVoterPaymentsModule()`. `emergencyDeallocate()` while open. On every entrypoint bound to the Relay, it chooses the router, the route and `minAmountOut` of each swap, triggers the idle paths and returns leftover tokens with `sweepToRelay()`.
- Assumptions: The holder swaps honestly and sets safe minimum output amounts to reduce slippage. It stays live and processes deposits and withdrawals without long delays. It converts and compounds rewards promptly after they arrive, follows the compound weight stored on the Hybrids, and does not time these calls to favor particular holders.
- Worst-case impact: Steals rewards before they are compounded or notified, by selling them through a pool it controls at a `minAmountOut` of one unit. Captures part of each reward by sandwiching a compound or convert with a deposit or withdrawal. Rotates to an authorized `VoterPaymentsModule` that refuses withdrawals, which freezes exits until a KEEPER rotates back. This requires `VPM_ADMIN_ROLE` to have authorized such a module.

VOTER_ROLE (on RelayBase)

- Trust level: Trusted.
- Permissions: `allocate()`, only while open. `collectAllocatorFees()` to any recipient.
- Assumptions: Allocates in the depositors' interest, to the gauges with the best return. Keeps enough weight free on chain 0 to pay queued exits, and deallocates in time when the queue grows, so the permissionless `close()` is never triggered by normal operation.
- Worst-case impact: Allocates the chain 0 float down to the withdrawal reserve, after which any holder can force a permanent `close()` by queueing an exit larger than the float and waiting `evacuationWindow`. Funds stay exitable. It can send allocator fees to any recipient address. Once replaced, its gauge votes in flight still apply until they expire. Its chain allocations in flight never expire.

RAKE_MANAGER (on RelayBase)

- Trust level: Partially trusted.
- Permissions: `proposeRakes()` and `executeRakes()`. A cut of both rates applies at once. Any raise waits `entrypointTimelock`. Each rate is capped by the implementation's `MAX_RAKE_PIPS`, below 100%. Can renounce its role.
- Assumptions: Sets rates that leave depositors a fair share of the rewards.
- Worst-case impact: Raises both rates to `MAX_RAKE_PIPS` after `entrypointTimelock`, which takes up to $1 - (1 - \text{MAX_RAKE_PIPS})^2$ of every later compound and notify. It cannot reach principal, accounted rewards or the rake already accrued.

RAKE_MANAGER_ADMIN (on RelayBase)

- Trust level: Partially trusted.

- Permissions: `setRakeManager()` to seat and remove rake managers, itself included, and `transferRakeManagerAdmin()`. Seated by the Relay Deployer at creation. Neither the Relay owner nor anyone else can remove it.
- Assumptions: Seats only rake managers that act in line with the rates communicated to depositors.
- Worst-case impact: Holds the worst case of `RAKE_MANAGER`, and cannot be unseated.

SWEEPER (on `ProtocolRelay`)

- Trust level: Trusted.
- Permissions: `sweep()`, only on a level 2 `ProtocolRelay`, to any recipient.
- Assumptions: Sweeps only to recipients that return the value to holders, such as the Relay's entrypoints, and never keeps it.
- Worst-case impact: Takes every ERC-20 balance above `accountedBalance`, which includes all incoming rewards before they are notified.

Leaf Operator (`operator` on `LeafVoter`)

- Trust level: Trusted.
- Permissions: `LeafVoter.claimRewards()` for the Relay's position on that chain, to any recipient, and `allocateGauges()` while local voting is enabled. Set and revoked per chain by the Relay owner through a cross-chain message. The Relay's configured `VoterPaymentsModule` can set the operator too: `Voter.setOperator()` only checks escrow authorization for the sAERO, and the Relay approves its module on that sAERO, so the module configures an operator directly and without `entrypointTimelock`. `RelayLeafModule.claim()` works only while the module is set as operator.
- Assumptions: The operator is the Relay's `RelayLeafModule` or another contract that claims only to the Relay's leaf recipient. It votes locally only in the depositors' interest.
- Worst-case impact: Claims every reward the Relay earns on that chain to an address it names. With local voting enabled, it also redirects the Relay's votes on that chain. Unrecoverable for rewards already claimed. Once the Relay owner replaces it, it keeps acting until the replacing message arrives. A stalled message that set it expires after the lifetime set by `VOTER_CONFIG_ROLE`.

Entrypoint Vetoer (`ENTRYPOINT_VETOER` on `ProtocolRelay`)

- Trust level: Partially trusted.
- Permissions: `vetoEntrypoint()` on any pending proposal and `transferVetoer()`. A Relay may be created with no vetoer.
- Assumptions: The vetoer is independent of the Relay owner, watches every entrypoint proposal, and vetoes only entrypoints that would harm depositors.
- Worst-case impact: Vetoes every entrypoint attachment on a level 2 `ProtocolRelay` forever, so a Relay without entrypoints never compounds or notifies. The owner cannot unseat it. The stranded rewards remain recoverable through `sweep()`.

Relay Deployer (`RELAY_DEPLOYER_ROLE` on `Voter`)

- Trust level: Trusted.
- Permissions: `createMaxiRelay()` and `createProtocolRelay()` on `RelayFactory`, choosing the initial roles, the initial rake rates and all parameters of the new Relay. The parameters are only checked to be non-zero, the seed and `minDeposit` to reach `MIN_STAKE_AMOUNT`, the rates to stay within `MAX_RAKE_PIPS` and the entrypoints to be bound to the new Relay. Apart from the rates and the name, they can never be changed. The role itself is granted and revoked by `CONFIG_ADMIN_ROLE` on the `Voter`.
- Assumptions: Initializes Relays with a sound configuration.

- Worst-case impact: Creates Relays with harmful roles or parameters. Depositors who check the configuration before depositing are not exposed, and existing Relays are unaffected.

4.1.2 Relay Module Roles

RelayLeafModule Owner (owner on RelayLeafModule)

- Trust level: Trusted.
- Permissions: Gives out, and thereby inherits the privileges of, the module's `KEEPER`. Can also call `rescueOutToken()` to send the whole `OUT_TOKEN` balance to a specified address.
- Assumptions: Relays only route leaf rewards to a module whose owner they trust.
- Worst-case impact: Steals the module's whole leaf balance with no delay: unsold rewards as `KEEPER`, sold rewards through `rescueOutToken()`.

RelayIcaModule Keeper (keeper on RelayIcaModule)

- Trust level: Trusted.
- Permissions: `dispatchPlan()` to swap and bridge interchain account balances to the Relay, choosing tokens, pools, `minAmountOut` and `targetRouter`. `sweepNative()` and `sweepToken()` over the module's root balance.
- Assumptions: Swaps at fair prices, stays available, and bridges proceeds to the Relay promptly.
- Worst-case impact: Steals nearly all unsold rewards in the interchain accounts by swapping through its own pools, and the module's root balance directly. Removed instantly by the Relay owner. Plans it dispatched stay executable after its removal until their unbounded `leafDeadline`, so removing it does not protect future rewards. Other stalled messages expire with their lifetime, such as claims to a replaced leaf recipient, a leaf operator that a later message superseded, or the votes of a replaced `VOTER_ROLE`.

RelayLeafModule Keeper (KEEPER on RelayLeafModule)

- Trust level: Trusted.
- Permissions: `swap()` with its own route and `minAmountOut`, `bridge()` with its own fee ceiling and `targetRouter`, `redeem()`, `sweepNative()`, and `sweepToken()` for any token except `OUT_TOKEN` and `RECEIPT_TOKEN`.
- Assumptions: Swaps at fair prices, bridges proceeds to the Relay promptly, and does not sweep unsold rewards.
- Worst-case impact: Steals every unsold reward, through `sweepToken()` or its own pools. Revoked instantly by the module owner.

4.1.3 Relay Tape Roles

RelayTape Owner (owner on RelayTape)

- Trust level: Trusted.
- Permissions: `setAllowedFactory()` adds and removes the factories that can write observations, with no delay.
- Assumptions: Only allowlists factories that forward reports solely from Relays they created, with the values those Relays report.
- Worst-case impact: Allowlisting an arbitrary address lets it write observations for any Relay and token, which corrupts the data an APR is computed from, including all later window reads across the running totals. Removing a factory stops recording for its Relays. No funds are at risk, since no Relay depends on the tape.

4.1.4 Roles Defined Outside Scope

These roles live on contracts outside the review scope but gate in-scope operations.

FEE_MANAGER and FEE_MANAGER_ADMIN_ROLE (on VoterPaymentsModule)

- Trust level: Fully trusted.
- Permissions: `setFee()` sets the deposit and withdrawal fees, up to 100%, as a default or per caller. `setRestricted()` limits a function to callers with their own fee entry, so a Relay without one can no longer call it. `removeFee()` deletes a caller's entry.
- Assumptions: Fees stay at their zero defaults and the withdrawal selector stays unrestricted. No in-scope code registers a Relay as a fee payer, so every Relay pays the default rate.
- Worst-case impact: Setting the default `withdrawToNFT()` fee just below 100% lets every exit from every Relay settle while nearly all of its weight goes to the fee accumulator. This is unrecoverable. Restricting the selector, or setting the fee to exactly 100%, instead freezes every exit, because the strict-order queue stops at a reverting head. The same lever on `depositIntoNFT()` takes all of a deposit above `minDeposit`.

VPM_ADMIN_ROLE (on VotingEscrow)

- Trust level: Fully trusted.
- Permissions: `grantRole()` and `revokeRole()` for `VPM_ROLE` on any address. `VPM_ROLE` lets a contract move stake between positions through `rebalanceUnderlying()`. A Relay only accepts a `VoterPaymentsModule` that holds it, and all deposits and exits run through that module.
- Assumptions: The configured `VPM` keeps the `VPM_ROLE` and the role is only granted to reviewed modules that keep the Relay accounting intact. A configured module uses the Relay's sAERO approval only to move weight, never to set a leaf operator of its own (see Leaf Operator).
- Worst-case impact: Revoking `VPM_ROLE` from the configured module freezes deposits and exits of every Relay using it, until the role is restored or each Relay's `KEEPER` switches to another module. Alone it cannot move a Relay's stake, since only the Relay's `KEEPER` can switch modules. Together with that `KEEPER` it steals the whole stake (see Collusion).

CHAIN_STATUS_ROLE , CONFIG_ADMIN_ROLE and GOVERNANCE_ROLE (on Voter)

- Trust level: Fully trusted.
- Permissions: `setChainStatus()` and `setEmergencyDeallocationAllowed()` for any registered chain other than chain 0. `CONFIG_ADMIN_ROLE` also administers the `Voter`'s operational roles, including `RELAY_DEPLOYER_ROLE`, and `GOVERNANCE_ROLE` administers `CONFIG_ADMIN_ROLE` in turn. Every role below them is therefore theirs to configure. On every leaf, the `LeafVoter`'s `CONFIG_ADMIN_ROLE` likewise administers `VOTER_CONFIG_ROLE`, which opens local voting and sets the allocation cooldown with no upper bound. It is fully trusted too. `GOVERNANCE_ROLE` also sets the `rakeTreasury` of `RelayFactory` with `setRakeTreasury()` and its relay tape and tape gas limit with `setRelayTape()`, with no delay. It also adds and removes relay factories in the `RelayFactoryRegistry` with `registerRelayFactory()` and `unregisterRelayFactory()`, with no delay. The Metarouter's `DEPOSIT_RELAY` command only accepts a Relay whose factory is in this registry.
- Assumptions: A chain is paused or suspended only in an emergency, emergency deallocation is enabled on a suspended chain that holds Relay weight, and the documented resume procedure is followed. The rake treasury is a protocol-controlled address. A relay factory stays registered as long as Relays it created accept deposits.
- Worst-case impact: Setting a chain to `Paused`, or to `Suspended` without allowing emergency deallocation, blocks both `evacuate()` and `emergencyDeallocate()` for weight booked there. Every Relay whose head exit exceeds its free chain 0 weight then has its whole queue frozen, open or closed. A leaf `VOTER_CONFIG_ROLE` has the same effect by setting an allocation cooldown so long that every `evacuate()` vote to that chain reverts with `CooldownActive`, or by setting `maxGauges` to zero. Reversible by the same role in one call. Repointing the rake treasury redirects the uncollected and future protocol rake of every Relay the factory created. A tape gas limit above what a transaction can provide makes every

`compound()` and `notifyReward()` of these Relays revert. Unregistering a relay factory makes `DEPOSIT_RELAY` revert with `UnauthorizedRelay` for every Relay that factory created. Users and integrations that deposit only through the Metarouter can then no longer deposit into these Relays until the factory is registered again. Direct `requestDeposit()` calls, pending deposits, exits and existing shares are unaffected.

VOTER_CONFIG_ROLE (on Voter)

- Trust level: Fully trusted.
- Permissions: `setMessageLifetime()` and `setAllocationLifetime()`, between 2 minutes and 365 days, with no delay. They set the expiry of claim, operator and gauge allocation messages.
- Assumptions: Lifetimes only cover normal cross-chain delivery, so stalled messages that claim to a replaced leaf recipient, set a leaf operator that a later message superseded or apply a replaced `VOTER_ROLE`'s votes expire soon after the rotation.
- Worst-case impact: With long lifetimes, such a stalled message still claims to the replaced leaf recipient, applies the replaced `VOTER_ROLE`'s votes, or overwrites the current operator with the superseded one, for up to a year.

4.1.5 End Users

Depositors and Share Holders

- Trust level: Untrusted.
- Permissions: `requestDeposit()`, naming any share recipient (only allow-listed ones on `ProtocolRelay`), `registerOnWithdrawQueue()`, `processWithdrawals()`, `processOverduePending()`, `extendLock()`, `claim()`, `claimRewards()`, `collectProtocolFees()`, `expressVote()`, `close()` once the head is uncovered, and `evacuate()` and `emergencyDeallocate()` once closed. Neither a deposit nor an exit can be canceled.
- Assumptions: Before depositing, users check the Relay's configuration on-chain and only use Relays whose roles they trust and whose parameters protect them: a bootstrap seed large enough that no deposit rounds to zero shares, short `keeperWindow` and `evacuationWindow`, an `entrypointTimelock` long enough to exit, an independent vetoer, and acceptable rake rates and `MAX_RAKE_PIPS`.

4.2 Collusion

The combinations below are the most severe ones, not a complete list.

`ENTRYPOINT_VETOER` is the only check on the owner inside a Relay. If they collude, or no vetoer exists, the owner can attach any entrypoint on a level 2 `ProtocolRelay` after `entrypointTimelock`, which then takes every un-notified balance.

A `KEEPER` together with `VPM_ADMIN_ROLE` could configure a malicious module and then rebalance the Relay's whole stake into a controlled NFT. This is the only combination that can steal all user funds.

4.3 External Dependencies

Governor and RelayVoteAdapter

- Role in system: Target of `expressVote()`, set from factory immutables and changeable by the owner through `setGovernor()`. The adapter is read-only and supplies the snapshot and the encoded cast.
- Worst-case impact if compromised: Casts the Relay's full voting weight in any direction. No funds are at risk.

Hyperlane

- Role in system: Transports every cross-chain message and token transfer: the Mailbox dispatches and delivers messages after the ISM verifies them, relayers deliver them, warp routes bridge tokens, and the interchain account router executes calls on remote chains.
- Assumptions: The Mailbox delivers each message at most once. The ISMs are correctly configured with an honest validator majority and are not replaced by permissive modules. At least one honest relayer is available per chain pair. Bridge rate limits are not reduced to zero. Each `RelayLeafModule`'s warp route stays live, since its `BRIDGE` is immutable and is the only way home for its `OUT_TOKEN`.
- Worst-case impact if compromised or unavailable: Forged messages mint tokens without a burn or execute arbitrary calls through an interchain account, taking the leaf rewards it holds. Without live relayers, delivery stalls, which can delay evacuation and exits until anyone runs a relayer. A warp route that stops working strands what its module holds on the leaf chain, which the module owner recovers with `rescueOutToken()` and the module `KEEPER` with `sweepToken()`, while the Relay owner repoints that chain's leaf recipient so later rewards avoid the module.

4.4 Asset Assumptions

`TOKEN` and the staked position

- Assumptions: `TOKEN` is `AERO`, a plain ERC-20 without transfer fees, rebasing or callbacks, where a maximum allowance never decreases. The Relay always owns its staked position.
- Worst-case impact: If the position leaves the Relay, exits revert while new deposits still flow into it. Unrecoverable.

Reward tokens

- Assumptions: Every registered reward token is a plain ERC-20 with no transfer fee, rebasing or callback. A reward token that can blacklist addresses never blacklists any contract of MetaDEX Relay. Registration only rejects the zero address, codeless addresses, the Relay and its two tokens.
- Worst-case impact: A token that moves less than stated drives `balanceOf` below `accountedBalance`, which makes `notifyReward()`, `pull()` and `sweep()` revert for that token, and the whole shortfall lands on the last claimant or fee collection. A token that blacklists the Relay makes every `claim()` of it revert. A blacklisted entrypoint or relay module strands the token there instead. The impact is confined to that token.

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 A Withdrawal May Deliver a Freshly Minted sAERO

System Consideration Version 1

In `ProtocolRelay`, `kick()` lets the owner drop an account from the allow-list and send its entire free position to the withdrawal queue. A holder who registers their own exit names the sAERO the weight should land in, but an eviction names no destination and always routes to a freshly minted sAERO sent to the evicted holder. Integrators must be aware that they can be handed one at any time.

A holder who does name a destination can be handed one too. The named sAERO is recorded at registration but only checked at settlement: `_routeWithdrawal()` re-reads both locks and, when the destination falls short of the Relay's own under the escrow's monotonic unlock rule, silently mints a fresh sAERO instead of reverting.

Integrators should treat the destination as a preference rather than a guarantee, and read the settled token ID from the `WithdrawProcessed` event instead of assuming the one they named. The fresh sAERO is minted without a receiver callback, so a contract that is handed one is never notified. A contract integrator that cannot enumerate or handle sAEROs it did not ask for should keep its destination permanent, or locked past the Relay's own unlock, for the whole time its exit is queued.

SC2 Compounding a Closed Relay

System Consideration Version 1

In `RelayBase`, `compound()` stays available after `close()`. The fees and incentives of the Relay's final votes keep arriving after closure. An exit on a closed Relay therefore burns only the holder's PT and leaves their YT, so that `notifyReward()` can still distribute these rewards to every final holder, regardless of when they exited. A compound bypasses this. It only raises the value of the PT still outstanding, so holders who exit early lose their share of every later compound to those who wait.

In particular, a Relay whose only entrypoint is a compounder cannot avoid this, since Maxi and level 1 Protocol Relays cannot add a converter later. Relay deployers should therefore always set a converter, and the `KEEPER` should convert rather than compound after closure. Users of a compounder-only Relay should expect a later exit to capture part of the rewards owed to earlier ones.

SC3 Security Considerations for Deployers of Maxi and Level 1 Relays

System Consideration Version 1

On a `MaxiRelay` or a level 1 `ProtocolRelay`, a token can only leave the Relay through an entrypoint swap into that entrypoint's fixed output token. `sweep()` exists only at level 2, the entrypoints are fixed at creation, and `notifyReward()` only distributes a registered token, while these tiers lock the registry after its first token is distributed. A token that no route on the approved Metarouter can swap into the output token can therefore be neither sold nor distributed. Such tokens do arrive: anyone can send tokens to the Relay, and a root reward claim delivers whatever fee and incentive tokens the voted gauges paid. Their value is lost to the holders who earned it.

Relay deployers should pick output tokens into which the fee and incentive tokens the Relay is expected to earn can be routed, and `VOTER_ROLE` should favor gauges whose rewards can be routed. A Relay that must be able to recover arbitrary tokens needs to be a level 2 `ProtocolRelay` with a `SWEEPER`.

SC4 Security Considerations for Parameter Selection and Forced Closure Through Spam

System Consideration Version 1

The only limits on how many entries anyone can add to a Relay's queues are the minimum deposit and exit sizes. With small minimums, the withdrawal queue is cheap to spam and costly to drain in its strict order. Since an exit's evacuation clock starts at registration, exits stuck behind spam, or behind a backlog during high gas prices, keep aging. Once one reaches the head older than `evacuationWindow` and the unallocated weight cannot cover it, anyone can close the Relay permanently.

Relay deployers should therefore choose these values with care, as none can be changed later:

- `evacuationWindow`: long enough for the `KEEPER` to drain a backlog and bring weight back, even at high gas prices.
- `minDeposit`: large enough that the yield a deposit forgoes while waiting for admission outweighs what an attacker gains by making others settle it. Every position can also queue one exit below `minWithdrawal` for its remainder, so `minDeposit` alone bounds how many such exits a given weight can create.
- `minWithdrawal`: large enough that one `minDeposit` yields at most one exit, which makes filling the withdrawal queue costly.

SC5 Idle TOKEN Rewards Cannot Be Split Between Compounding and Distribution

System Consideration Version 2

In `BaseEntrypoint`, `compoundIdleBalance()` compounds the Relay's whole unaccounted `TOKEN` balance. `convertIdleBalance()` distributes the Relay's whole unaccounted balance of the target token to YT holders.

If `TOKEN` is also a registered reward token, the `KEEPER` cannot leave part of the idle `TOKEN` to be compounded later. Every idle conversion also distributes the `TOKEN` waiting to be compounded, and every idle compound also stakes the `TOKEN` waiting to be distributed.

SC6 Security Considerations for Relay Tape Consumers

System Consideration Version 4

In `RelayTape`, `_observeStored()` picks the ring from the lookback timestamp alone. Lookbacks up to `HOURLY_SPAN` (48 hours) read only the hourly ring and longer ones read only the daily ring. The other ring is never consulted, which has the following implications:

1. Beyond 48 hours, the daily ring answers with at most one observation per day, even if the hourly ring holds an older and closer one. The hourly ring has 48 slots and is filled per write, not per hour. With sparse feeds it can therefore reach back further than 48 hours.
2. Within 48 hours, once the hourly ring has wrapped, its oldest observation can be newer than the target. `_observeSingle()` then returns that observation, even if the daily ring holds one at or before the target that is closer to it.

3. The far edge can sit on either side of the requested time, and nothing in the return value signals it. A consumer must derive rates from the returned `blockTimestamp` and not from the requested lookback.
4. The resolution changes at 48 hours. Lookbacks of 48 hours and 48 hours plus one second can resolve to observations up to a day apart. Window-based figures such as an APR can therefore jump at this boundary.

SC7 Tape Readings Before the First Observation

System Consideration Version 4

In RelayTape, a zero lookback in `observeReward()` and `observeBacking()` returns the live reading, made of the tape's running total and the relay's current reward index or backing per share. Until a series stores its first observation, every entry, including the live one, is zeroed. This holds even if the relay already has a non-zero reward index or backing, for example because the tape was attached after the relay started.

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

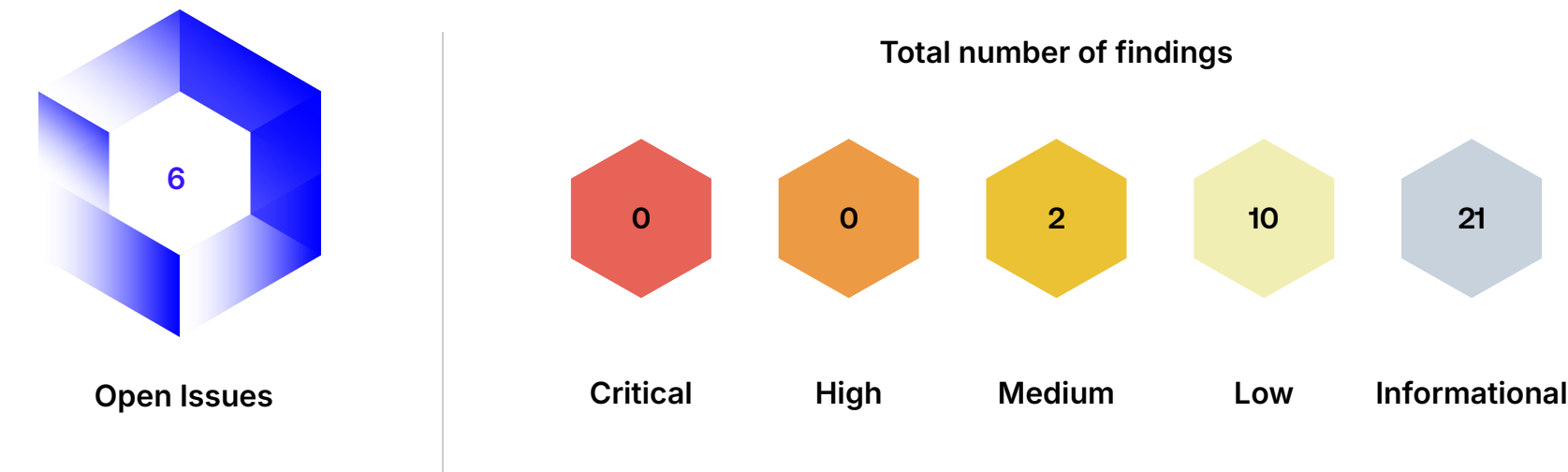
We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

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.



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. Non-informational findings with any other status, including partial fixes, acknowledgements, and accepted risks, remain open.

Medium	#001	Insufficient Protection Against Inflation Attacks	Code Corrected	✓
Medium	#002	Leaf Operator Can Block the Return of a Relay's Voting Weight	Risk Accepted	⚠
Low	#003	Anyone Can Force a Holder's Reward to Round Down	Risk Accepted	⚠
Low	#004	Canceling a Proposed Leaf Recipient Also Disables Claims for That Chain	Code Corrected	✓
Low	#006	Keeper Can Sweep the Output Token on Arc	Code Corrected	✓
Low	#010	Double Entry Point Tokens Bypass the Bound on Owed Rewards	Risk Accepted	⚠
Low	#013	Grow Budget Is Imprecise	Code Corrected	✓
Low	#015	ICA Module Plan Bricks with Some Weird ERC-20 Tokens	Code Corrected	✓
Low	#018	Intermediate Tokens from a Swap Batch Are Left Behind on the Entrypoint	Code Corrected	✓
Low	#026	Repeated Partial Reward Claims Leave Dust Behind	Risk Accepted	⚠
Low	#031	Hyperlane Router Changes Can Strand Leaf Rewards	Risk Accepted	⚠
Low	#032	Removed Keeper's Dispatched Plans Stay Executable	Risk Accepted	⚠
Info	#005	Interchain Account View Returns Addresses the Module Can Never Use	Code Corrected	✓
Info	#007	_extendLock Is Virtual but Never Overridden	Code Corrected	✓
Info	#008	An Already Processed Deposit Reverts an Entire Overdue Batch	Code Corrected	✓
Info	#009	An Oversized Entrypoint Timelock Permanently Disables the Proposal Flows	Acknowledged	⊖
Info	#011	Ejecting a Holder Settles Their Rewards Twice	Code Corrected	✓
Info	#012	Exiting to Destination tokenId Zero Burns the Holder's Whole Withdrawal	Specification Changed	✓
Info	#014	Holders Can Time Inflows to Steer Reward Distribution	Acknowledged	⊖
Info	#016	Inconsistent Error Usage for NoPendingDeposits	Code Corrected	✓
Info	#017	Incorrect NatSpec Comment on Total Backing	Code Corrected	✓

Info	#019	Leaf Recipients Can Be Configured for Root	Code Corrected	✓
Info	#020	Liveness of Withdrawals Relies on Falsely Documented Escrow Behavior	Specification Changed	✓
Info	#021	RelayFactory Configuration Is Only Partly Validated	Acknowledged	⊖
Info	#022	Missing Sanity Check on RelayIcaModule	Code Corrected	✓
Info	#023	Mutable Payments Module Is Named as Though It Were Immutable	Code Corrected	✓
Info	#024	A Switched Payment Module Can Block All Withdrawals	Acknowledged	⊖
Info	#025	Removing Principal Token Voting Power Can Be Undone by Anyone	Acknowledged	⊖
Info	#027	Several State Changes Emit No Event	Code Part. Corrected / Acknowledged	⏮
Info	#028	Tokens Left on an Entrypoint Can Be Swept by Anyone	Code Corrected	✓
Info	#029	Undocumented Remapping of the Withdrawal Recipient	Specification Changed	✓
Info	#030	Identical Backing Getters Are Documented Differently	Code Corrected	✓
Info	#033	RelayLeafModule Cannot Bridge Partial Amounts	Acknowledged	⊖

7.2 Descriptions

#001 Insufficient Protection Against Inflation Attacks

Trust

Medium

Version 1

Code Corrected 

In `QueueLib`, an admitted deposit is priced as

```
uint256 _shares = _supply == 0 || _backing == 0 ? _entry.amount : (uint256(_entry.amount) *  
    _supply) / _backing;
```

with no virtual shares or dead shares as protection against inflation attacks. Instead, the only protection against a manipulated price is the seed amount of tokens that `initialize()` mints at 1:1 to `bootstrapOwner`. Its size is only checked to be non-zero.

An attacker can inflate the share price by sending `TOKEN` or weight to the Relay. This only raises `totalBacking` once the `KEEPER` books it through an entrypoint's `compoundIdleBalance()` or through `processDonations()`. If that happens after a victim requests a deposit but before it is admitted, the deposit receives zero shares and its weight goes to the existing holders. The attack therefore needs the `KEEPER` to book the donation first, either in collusion or while leaving the deposit pending past `keeperWindow`, after which anyone can admit it with `processOverduePending()`. It is profitable for the bootstrap owner, who can first exit all but one share and then recover the full donation, and for anyone else when the seed is small. For example, with a seed of 1000 shares, rounding down a `1e18` deposit requires a donation of about `1000 * 1e18` to the bootstrap owner, so the attacker only breaks even after attacking at least 1000 such deposits. The Relay deployer names both the bootstrap owner and the `KEEPER`.

Code Corrected

The seed can no longer leave an open Relay. `initialize()` now mints the seed on the principal token only, and `QueueLib.registerExit()` requires the exiting holder of an open Relay to cover the exit with yield tokens as well. Since the yield-token supply never includes the seed, the principal supply cannot fall below the seed while the Relay is open. Further, the seed must now be at least `MIN_STAKE_AMOUNT` (`1e18`), and so must `minDeposit`. Finally, `processDonations()` was removed, so donated weight is never booked into `totalBacking`.

`TOKEN` sent directly to the Relay can still be staked into `totalBacking` by the `COMPOUNDER` through `compound()`. With at least `1e18` shares outstanding, however, rounding an admitted `1e18` deposit down to zero shares requires a donation above `1e18 * 1e18 = 1e36` base units. That is `1e18` AERO, far more than the total AERO supply, so that much does not exist.

After the Relay is closed, the bootstrap owner can exit the seed. In collusion with the `COMPOUNDER`, it could register an exit of all but `minWithdrawal` shares, have a donation booked through `compound()`, let queued deposits round to zero and withdraw the donation together with the deposits. The attack is bounded the same way, because the principal supply cannot fall below `minWithdrawal` and the share price starts at one and never falls. Rounding a deposit to zero then requires a donation of more than the deposit times `minWithdrawal`. Protection in this case therefore depends on the `minWithdrawal` the Relay deployer sets, which is only required to be non-zero.

#002 Leaf Operator Can Block the Return of a Relay's Voting Weight

Trust

Medium

Version 1

Risk Accepted 

In `RelayBase`, `evacuate()` brings the Relay's voting weight back from a leaf chain by sending that chain a vote that returns the whole booked amount. The leaf enforces a cooldown between gauge votes, and every vote resets it, so on a chain where governance has enabled local voting, the Relay's operator can reset the

cooldown and make each incoming return vote fail. The weight stays booked on the leaf and the exits stall indefinitely.

The Relay owner can remove an operator at any time with `revokeOperator()`. Relying on this makes the guarantee that user principal cannot get permanently stuck depend on the Relay owner, whose trust level is Trusted, not Fully trusted. Governance can also end the block, as `VOTER_CONFIG_ROLE` can disable local voting on the leaf and `CHAIN_STATUS_ROLE` can suspend the chain and allow emergency deallocation.

Risk Accepted

Dromos Labs plans not to use a leaf operator with local voting and states:

Acknowledged. In our intended setups, the leaf operator will not be able to initiate local allocations.

#003 Anyone Can Force a Holder's Reward to Round Down

Security

Low

Version 1

Risk Accepted 

An incoming reward is distributed per unit of yield-token supply in `RelayRewardsLib.notifyReward()`:

```
uint256 _indexDelta = (_amount * _accScale) / _supply;
```

and later credited to each individual holder in `RelayRewardsLib._settleAll()`:

```
uint256 _delta = _index - _userCheckpoint[_holder][_token];
if (_delta > 0) {
    _pendingReward[_holder][_token] += (_balance * _delta) / _accScale;
    _userCheckpoint[_holder][_token] = _index;
}
```

The index is in `accScale` precision, but `_pendingReward` is in the token's own decimals, so a settlement can lose up to one token unit. Settling a user right after a reward distribution credits approximately their share of the distribution:

$$\text{credit} = \left\lfloor \frac{\text{balance} \cdot \left\lfloor \frac{\text{amount} \cdot \text{accScale}}{\text{supply}} \right\rfloor}{\text{accScale}} \right\rfloor \approx \frac{\text{amount} \cdot \text{balance}}{\text{supply}}$$

which rounds to zero as long as it stays below one token unit.

A single token unit can be economically significant for a high-value, low-decimal reward token. WBTC has 8 decimals, so one unit is approximately 0.001 USD at the time of writing, and GUSD has only 2 decimals, so one unit is worth 0.01 USD.

In combination with a permissionless settlement process, this presents a griefing attack vector. Note that the rounded-down amount is paid to nobody else and stays locked in `accountedBalance`. The yield-token hook settles the recipient of every mint, so anyone can force a settlement and have a holder's rewards rounded down on any open Relay by calling `requestDeposit()` with the holder as the recipient, at a cost of at least `minDeposit` that the holder keeps. The shares mint when the deposit is processed, which anyone can trigger once `keeperWindow` has passed. On a Protocol Relay, this reaches every allow-listed holder. On a Maxi Relay created with a transferable yield token, the transfer hook settles the recipient as well, so sending one unit of YT to a holder suffices.

Further, any integrating contract that can be made to call `RelayBase.claim()` (for example, by depositing into a vault contract) can be force-settled and thereby rounded down, as `claim()` first settles the user.

Risk Accepted

Dromos Labs accepts the risk. The NatSpec of `IRelay.claim()` now documents that each settle rounds the holder's credit down to the token's smallest unit, that the remainder stays in `accountedBalance` with no owner, and that third parties can trigger a settle with a transfer or a deposit to the holder.

#004 Canceling a Proposed Leaf Recipient Also Disables Claims for That Chain

Design

Low

Version 1

Code Corrected ✓

In `RelayBase`, changing the address a leaf chain's rewards are claimed to takes `entrypointTimelock`, and `clearLeafRecipient()` both cancels a pending proposal and removes the current address. With no address set, the permissionless `claimRewards()` reverts for that chain, so rewards keep accruing on the leaf and cannot be claimed permissionlessly until a new address is proposed and executed a full timelock later. This also removes what makes the timelock useful, which is that anyone can claim the rewards to the current address before a new one takes effect.

Code Corrected

`clearLeafRecipient()` now only removes the current recipient, and the new `cancelLeafRecipient()` only drops the pending proposal. The owner can therefore cancel a proposal while claims to the current recipient keep working.

#006 Keeper Can Sweep the Output Token on Arc

Trust

Low

Version 1

Code Corrected ✓

In `RelayLeafModule`, `sweepToken()` refuses `OUT_TOKEN`, so the keeper cannot sweep proceeds that are meant to be bridged to the Relay. `sweepNative()`, however, sends the module's whole native balance to any address the keeper names. On Arc, the native currency USDC also has an ERC-20 entry point that shares the same balance. On a module with USDC as its `OUT_TOKEN`, the keeper can therefore take all proceeds as native currency, bypassing the restriction on `sweepToken()`.

Code Corrected

`RelayLeafModule.sweepNative()` now records the module's `OUT_TOKEN` balance before the sweep and reverts with `CannotSweepOutToken` if that balance decreased after the native transfer.

#010 Double Entry Point Tokens Bypass the Bound on Owed Rewards

Trust

Low

Version 1

Risk Accepted ⚠

In `RelayRewardsLib`, `pull()`, `sweep()`, `compound()` and `notifyReward()` can only pull a token's balance above `_accountedBalance`, as anything below is already owed to YT holders. That bound is kept per token address. A double-entry-point token has two addresses that share one balance.

As such, an entrypoint pulling for the second entry point, or a `SWEEPER` sweeping it, can move rewards that are already owed to holders. If both addresses are registered as reward tokens, `notifyReward()` can also promise the same balance twice. Once the balance falls below the owed amount, the bound for the first address underflows and reverts:

```
uint256 _unaccounted = IERC20(_token).balanceOf(address(this)) - _accountedBalance[_token];
```

In either case the Relay owes more of the token than it holds, and the last holders to claim cannot be paid.

Risk Accepted

Dromos Labs accepts the risks.

#013 Grow Budget Is Imprecise

Correctness

Low

Version 1

Code Corrected ✓

In `AllocationLib`, `allocate()` limits how much weight can be moved off chain 0 by deducting the pending withdrawal reserve from the free chain 0 weight:

```
uint256 _claimable = uint256(_relay.VOTING_ESCROW().staked(_tokenId).amount) -
    _relay.pendingDepositWeight();
uint256 _reserve = _supply == 0 ? 0 : (_relay.pendingWithdrawalShares() * _claimable) /
    _supply;
uint256 _free = _voter.allocationChainAmounts(_tokenId, _CHAIN0);
uint256 _growBudget = _free > _reserve ? _free - _reserve : 0;
```

However, the reserve is prone to change. At any point in time a user could register a new withdrawal, raising `pendingWithdrawalShares` with nothing set aside for it. As a result, `VOTER_ROLE` would have to deallocate quickly. Further, `QueueLib` rounds the minted shares down to $(_entry.amount * _supply) / _backing$ but adds the whole `_entry.amount` to `totalBacking`, so every admitted deposit raises the backing per share slightly. Exits are paid at the price at the time they are processed, so a queued exit can grow a few wei above the `_reserve` set aside. If a vote spent the whole grow budget, the drain stops at that exit, and if it stays stuck for `evacuationWindow`, anyone can close the Relay.

As long as `VOTER_ROLE` leaves some headroom above `_reserve`, this does not cause an issue.

Code Corrected

`AllocationLib.allocate()` now prices the withdrawal reserve against the Relay's whole stake, including the weight parked for queued deposits and any donated weight, instead of the stake net of pending deposits. Admitting a deposit raises the share price by at most the deposit amount over the supply, which is the case where it mints zero shares. As such, including the full stake is an assumption conservative enough that it cannot be exceeded.

Withdrawals registered after an allocation still raise `pendingWithdrawalShares` with nothing set aside for them, so `VOTER_ROLE` still needs to deallocate in time for such exits.

#015 ICA Module Plan Bricks with Some Weird ERC-20 Tokens

Correctness

Low

Version 1

Code Corrected ✓

`RelayIcaModule` sells gauge fee and incentive rewards on the leaf through an interchain account that executes a plan the module builds. The plan approves the Metarouter once per token sold with a raw `IERC20.approve`, then runs a batch of swaps that pull a fixed `amountIn` from the ICA. Rewards can be paid in any ERC-20, and some non-standard tokens can make the plan revert.

Some tokens, such as USDT or KNC, do not allow approving a non-zero amount while a non-zero allowance is already set. A CL exact-in swap can consume less than `amountIn` if the pool runs out of liquidity and the output amount still meets `minAmountOut`, so some allowance remains with the Metarouter. The account never resets it, so every later `approve` call with such a token reverts, blocking future swaps.

No funds are permanently stranded. The Relay owner can replace the Metarouter in the `LeafConfig`, and the new Metarouter starts from a zero allowance. The same fix can be used for reward tokens that the current Metarouter does not support, such as trading fee-on-transfer tokens on CL routes.

Code Corrected

`RelayIcaModule` now appends a zero approval to the Metarouter for every token sold after the swap batch of each plan, so no allowance carries over to the next plan. Tokens that revert on a zero approval cannot be sold through a plan, which is documented in `IRelayIcaModule`.

#018 Intermediate Tokens from a Swap Batch Are Left Behind on the Entrypoint

Correctness Low Version 1

Code Corrected ✓

In `BaseEntrypoint`, `_pullSwapAndValidate()` hands the Relay only two assets: the measured output delta and the whole `tokenIn` balance. The Metarouter returns more than that, because it sweeps every token it tracked during the batch back to its caller, which is the entrypoint. A keeper-supplied batch can therefore leave a third token behind, such as an intermediate hop token routed back to the router or a funded input that a pips-denominated spend only partly consumed, and nothing in the entrypoint forwards it. No entrypoint has a sweep, so that value sits outside the Relay's accounting until someone names the token as `tokenIn` on a later call, which is the unauthenticated path described in [Tokens Left on an Entrypoint Can Be Swept by Anyone](#).

Code Corrected

The new keeper-only `sweepToRelay()` in `BaseEntrypoint` has been added in `Version 2` to sweep any stuck tokens to the bound Relay. Further, in `Version 2` entry points are bound to a given relay and can no longer be swept by anyone.

#026 Repeated Partial Reward Claims Leave Dust Behind

Design Low Version 1

Risk Accepted ⚠

In `RelayBase`, `claimRewards()` and, on a leaf chain, `claim()` in `RelayLeafModule` let anyone claim the Relay's fees and incentives at any time. Claiming in many small steps means the rewards are rounded down repeatedly instead of just once.

Risk Accepted

Dromos Labs has acknowledged the risk, but has decided to keep the code unchanged.

#031 Hyperlane Router Changes Can Strand Leaf Rewards

Design

Low

Version 1

Risk Accepted 

In `RelayIcaModule`, `dispatchPlan()` and `interchainAccount()` pass `address(0)` for both the ICA router and the ISM in the `Metarouter`'s `IcaConfig`. For that case the `Metarouter` resolves both from the ICA router's own registries, `routers(domain)` and `isms(domain)`. The interchain account the module derives and controls on the leaf is a function of those two addresses.

The Hyperlane owner can change that registration at any time, for example through `InterchainAccountRouter.enrollRemoteRouterAndIsms()`. If the domain is re-registered with a different router or ISM, the `RelayIcaModule` will derive and control a new, empty ICA on the leaf chain and the rewards already claimed into the old one can no longer be accessed, as the keeper cannot pass any other router or ISM. Further, if the owner instead unenrolls the domain, every plan dispatched reverts.

Risk Accepted

Dromos Labs accepts the risk and states:

We investigated this and confirmed with the Hyperlane team that the registered `routers(...)` and `ism[...]` values are not expected to change during the normal lifecycle of the contract. We will therefore acknowledge the finding.

#032 Removed Keeper's Dispatched Plans Stay Executable

Trust

Low

Version 1

Risk Accepted 

In `RelayIcaModule`, the keeper calls `dispatchPlan()` to send a plan to the interchain account on a leaf chain. Each plan carries swap legs whose pools and `minAmountOut` the keeper chooses. A keeper can dispatch any number of plans before the owner removes it, and those plans cannot be canceled once dispatched.

Each plan stays executable until its `leafDeadline`, which the keeper sets without any upper bound. Anyone can execute a plan using the salt and calls published in `PlanDispatched`. An execution that fails for lack of funds can be retried later. A keeper can therefore arm a set of never-expiring plans with fixed amounts, a `minAmountOut` of one unit and pools it controls. After its removal, it can execute them against every reward batch that later reaches the interchain account. The owner has no way to stop this, so removing the keeper does not protect future rewards.

Risk Accepted

Dromos Labs accepts the risk and states:

The relay could start directing rewards to a different leaf recipient. Rewards already in the old recipient, or in flight via cross-chain claims, would still be at risk (note that cross-chain claims have an expiry).

#005 Interchain Account View Returns Addresses the Module Can Never Use

Informational Version 1

Code Corrected ✓

In `RelayIcaModule`, every plan the module dispatches leaves the destination router and security module unset, so the interchain account it commands is always the one Hyperlane derives for the Metarouter from the router and security module registered for the target chain. The `interchainAccount()` overload taking a domain returns that account. However, the overload `interchainAccount(address _router, address _ism)` taking a router and a security module returns an account derived from whatever pair the caller names, including modules that cannot be reached by a plan.

Code Corrected

The `interchainAccount(address _router, address _ism)` overload was removed.

#007 `_extendLock` Is Virtual but Never Overridden

Informational Version 1

Code Corrected ✓

In `RelayBase`, `_extendLock()` is declared as `virtual`, but no contract in scope overrides it.

Code Corrected

`_extendLock()` in `RelayBase` is no longer declared `virtual`.

#008 An Already Processed Deposit Reverts an Entire Overdue Batch

Informational Version 1

Code Corrected ✓

In `QueueLib`, `processOverdueDeposits()` lets anyone process named pending deposits by ID once they have aged past `keeperWindow`. It reverts the entire call when any named entry was already processed, while the `KEEPER`'s ordered walk `processDeposits()` skips such entries instead. Another caller or the `KEEPER` can therefore consume one named entry first and make the whole batch revert, wasting the caller's gas and forcing a resubmission with a narrower list.

Code Corrected

`QueueLib.processOverdueDeposits()` now skips named entries that were already processed and reverts with `DepositAlreadyProcessed` only if no entry of the batch was processed.

#009 An Oversized Entrypoint Timelock Permanently Disables the Proposal Flows

Informational Version 1

Acknowledged 

In `RelayInitLib`, `setUp()` checks `entrypointTimelock` only for being non-zero and stores it with no upper bound. Nothing can change the value afterwards, so a Relay created with an oversized one can never configure a leaf claim recipient, seat a leaf operator, or attach an entrypoint on a Protocol Relay that started at level 2. Further, a value above `type(uint256).max - block.timestamp` makes every propose call revert with an unlabeled arithmetic panic (0x11) rather than the `InvalidEntrypointTimelock` error that exists for exactly this class of misconfiguration.

Acknowledged

Dromos Labs acknowledged the issue.

#011 Ejecting a Holder Settles Their Rewards Twice

Informational Version 1

Code Corrected 

In `RelayRewardsLib`, `ejectHolder()` settles the holder's accrual across every registered reward token and then queues their whole free position for exit. Queueing moves no shares, and the settle has already advanced the holder's checkpoints to the current index. While the Relay is open, draining that queue entry burns the holder's yield tokens, and the burn calls back into the Relay's settle hook, which settles the same holder again at the same balance. The eviction therefore pays for a full pass over the reward-token registry that credits nothing the drain would not credit on its own, and splitting one accrual into two floored divisions can drop a further wei per reward token.

Code Corrected

The reward settlement was removed from the eviction. The rewards are settled when the drain burns the yield tokens, or on claim.

#012 Exiting to Destination tokenId Zero Burns the Holder's Whole Withdrawal

Informational Version 1

Specification Changed 

In `QueueLib`, `registerExit()` refuses only the Relay's own sAERO and destinations too wide for the stored field, so a holder can register an exit to token ID 0, which in `VotingEscrow` is not an sAERO but the protocol's burn accumulator. The fallback that reroutes an unusable destination to a freshly minted sAERO does not catch this one, because the accumulator reads as a permanent position once it has been credited, so the drain settles the exit normally and the holder's weight is lost.

Specification Changed

Dromos Labs documents the aforementioned behavior as intentional in the NatSpec of `IRelay.processWithdrawals()`.

#014 Holders Can Time Inflows to Steer Reward Distribution

Informational Version 1

Acknowledged 

In `BaseEntrypoint`, `convertIdleBalance()` and `compoundIdleBalance()` take no amount and instead distribute the Relay's whole unaccounted balance of a given token. The amount in the Relay at the moment of the call can be influenced by a user.

Anyone can deliver a pending Hyperlane transfer on the root chain, which mints the bridged tokens to the Relay. A PT or YT holder can therefore make a large batch land right before the call that favors them, since a convert pays YT holders and a compound pays PT holders. The open `bridge()` of `RelayLeafModule` makes this easier, as anyone can send a module's balance to the Relay once it exceeds the minimum bridge amount. Every token that arrives before a convert or compound is distributed by it. On a `MaxiRelay` with a transferable YT, one could also buy YT right before a convert and sell it afterwards to take a share of rewards earned before the purchase.

Acknowledged

Dromos Labs acknowledged the issue, but has decided to keep the code unchanged.

#016 Inconsistent Error Usage for NoPendingDeposits

Informational Version 1

Code Corrected 

In `QueueLib`, the function `processDeposits()` reverts with `NoPendingDeposits` when the queue holds no live entry. `processOverdueDeposits()` reverts with the same error when the caller supplies an empty list for `_ids`, a condition that says nothing about the queue and triggers even while entries are waiting.

Note that in the case of an empty `_list`, the function later reverts with `DepositNotFound`, as an empty queue can never contain an ID:

```
if (!_queueCache.exists(_id)) revert IRelay.DepositNotFound();
```

Code Corrected

`QueueLib.processOverdueDeposits()` now reverts with the dedicated `EmptyDepositIds` error when called with an empty `_ids` array.

#017 Incorrect NatSpec Comment on Total Backing

Informational Version 1

Code Corrected 

In `IRelay`, `totalBacking()` is documented as:

```
/// @notice TOKEN-denominated weight backing the current shares: admitted deposits plus
///         compounds
///         minus withdrawals. An internal counter, never read from `balanceOf` or
///         `VE.staked()`,
///         so a donation cannot inflate the share price.
```

Donations are missing from this description. `processDonations()` increments `totalBacking` by the `TOKEN` amount donated directly into the Relay's sAERO.

Code Corrected

`processDonations()` was removed, so donations no longer increase `totalBacking` and the NatSpec of `IRelay.totalBacking()` now matches the code.

#019 Leaf Recipients Can Be Configured for Root

Informational Version 1

Code Corrected ✓

In `RelayBase`, `proposeLeafRecipient()` and `executeLeafRecipient()` accept the local chain ID, but `RelayRewardsLib.recipientFor()` resolves that case to the Relay itself, so the stored entry never takes effect.

Code Corrected

`proposeLeafRecipient()` now reverts with `LeafRecipientOnRoot` for the Relay's local chain ID.

#020 Liveness of Withdrawals Relies on Falsely Documented Escrow Behavior

Informational Version 1

Specification Changed ✓

A user can choose a destination NFT for their withdrawn weight. The function `QueueLib._routeWithdrawal()` indirectly checks the existence of this ID in `_destinationFallsShort` (a missing ID reads as an all-zero stake and so is never long enough) and later compares the destination against the source's lock. If these checks fail, a new token ID is created instead. These checks are critical to prevent reverts in the `VoterPaymentsModule` and `VotingEscrow`, and by extension a permanent deadlock of the withdrawal queue, as `RelayBase` cannot handle reverts gracefully:

```
// QueueLib, the drain's check: the destination's live lock against the source's.
function _destinationFallsShort(...) private view returns (bool _short) {
    IVotingEscrow.StakedBalance memory _source = _votingEscrow.staked(_relayTokenId);
    IVotingEscrow.StakedBalance memory _dst = _votingEscrow.staked(_destination);
    _short = _source.isPermanent ? !_dst.isPermanent : (!_dst.isPermanent && _dst.end <
        _source.end);
}
```

Later, in the `VotingEscrow`, the function `AllocationLogicLibrary._addChain0Contribution()` also checks that, for a destination that already holds a chain 0 allocation, the shape the `Voter` recorded for that NFT matches the NFT's live shape. Otherwise, it reverts with `DstShapeStale`:

```
// AllocationLogicLibrary._addChain0Contribution, via _shapeDiffers: the destination's
// recorded shape against its live one.
function _shapeDiffers(...) private pure returns (bool _differs) {
    _differs = _stakeEndA != _stakeEndB || _isPermanentA != _isPermanentB;
}
```

The first check does not imply the second check. The second check never fires, since the `VotingEscrow` re-anchors the stake on every call path that reshapes it. `upgradeToPermanentStake()`, `downgradeFromPermanentStake()` and `createStake()` / `reviveStake()` / `increaseStakeAmount()` /

`increaseStakingPeriod()` via `_depositFor()` each call `VOTER.parkOnChain0()`. That function re-anchors a stale stored shape before booking the tokens to chain 0, including when there is nothing to park. The remaining shape-changing path, `withdraw()`, reaches `VOTER.clearToken()` instead, which deletes the `Voter`'s record for that token outright, so it leaves nothing stale either.

The NatSpec of `IVotingEscrow.increaseStakingPeriod()` and `upgradeToPermanentStake()` states the opposite. Both say they change the stake "without touching the Voter's chain0 mirror" and that a previously-voted token must re-vote afterwards, "otherwise inbound VPM rebalances into it revert `IVoter.DstShapeStale` until the re-vote".

Were that description accurate, the Relay would inherit a high-severity issue. An attacker would vote with an NFT, register an exit naming it, then make it permanent without voting again. That single action passes `_destinationFallsShort()`, since a permanent destination is never short, while leaving the recorded shape stale for the second check. Withdrawals are processed only from the head of the queue, and no fallback exists for a destination that reverts, so the queued shares of every user would remain locked indefinitely.

The Relay is therefore safe only because of the escrow's behavior. The escrow's own interface documentation is incorrect, and the security of the Relay relies on escrow behavior that contradicts its documentation.

Specification Changed

The NatSpec of `IVotingEscrow.increaseStakingPeriod()` and `upgradeToPermanentStake()` now states that both re-anchor the stake shape recorded in the Voter through `IVoter.parkOnChain0()`, so no re-vote is needed. This matches the implementation.

#021 RelayFactory Configuration Is Only Partly Validated

Informational Version 1

Acknowledged 

In `RelayFactory`, the constructor checks only that its addresses are non-zero and that the two tier implementations carry code. It does not check that:

1. the `VPM` holds the escrow's `VPM_ROLE`. `RelayInitLib` requires it, so an unauthorized module reverts every relay deployment.
2. `VOTING_ESCROW` and `VOTER` match the implementations'. A `VOTING_ESCROW` mismatch reverts every creation, since the clone verifies seed ownership on its own escrow. A `VOTER` mismatch survives, because nothing reads it during creation, and the Relay then allocates, evacuates and claims through a `Voter` other than the one whose deployer role authorized it.
3. each implementation is the tier it is filed under. `relayType()` is callable on both, but the factory reads neither and hardcodes the emitted type.

Nor can any of them be corrected later, as the factory holds them as immutables with no setter. `VPM_ROLE` is revocable, so retiring a module that live Relays simply rotate away from also ends creation on this factory. The only repair is a new factory, which starts with an empty `isRelay()` registry and needs a new `Metarouter` alongside it, since its `RELAY_FACTORY` is immutable as well.

Acknowledged

Dromos Labs acknowledged the finding.

#022 Missing Sanity Check on RelayIcaModule

Informational Version 1

Code Corrected ✓

In `RelayIcaModule`, `ROOT_DOMAIN` names the domain every plan's bridge leg pays out on, and is fixed at construction. It is only checked to be non-zero. Neither the module nor `RelayIcaModuleDeployer` checks it against the `localDomain()` the interchain account router exposes for the chain they are deployed on.

Code Corrected

`RelayIcaModuleDeployer` no longer takes the root domain as a constructor argument. It now sets `ROOT_DOMAIN` to the `localDomain()` of the `Metarouter`'s interchain account router and passes it to every `RelayIcaModule` it deploys.

The constructor of `RelayIcaModule` still accepts any non-zero `_rootDomain`, so only modules deployed through `RelayIcaModuleDeployer` carry this guarantee.

#023 Mutable Payments Module Is Named as Though It Were Immutable

Informational Version 1

Code Corrected ✓

In `RelayBase`, `VPM` is a mutable storage variable, but it is written in capital letters, which the style guide reserves for constants and immutables.

Code Corrected

The mutable `VPM` storage variable of `RelayBase` and its getter in `IRelay` were renamed to `vpm`.

#024 A Switched Payment Module Can Block All Withdrawals

Informational Version 1

Acknowledged ☹

In `RelayBase`, `processWithdrawals()` pays queued exits strictly in FIFO order. No function cancels or skips a registered exit, so a single exit whose payout always reverts blocks every exit behind it. Each payout is a call to the `VoterPaymentsModule`:

```
vpm.withdrawToNFT(_relayTokenId, _destinations);
```

`QueueLib` avoids the reverts the current module can raise on a holder's exit. For example, `_destinationFallsShort()` detects a destination NFT the escrow would refuse and pays the exit into a fresh NFT instead. These guards are part of the Relay's logic, which cannot be upgraded. The module is not fixed: with `setVoterPaymentsModule()`, the `KEEPER` can switch the Relay to any other module the `VotingEscrow` authorizes. Nothing ensures that a new module only raises the reverts `QueueLib` anticipates, and a module that rejects a single exit would strand all user funds in the Relay.

Note that one revert that `QueueLib` does not avoid is a withdrawal fee of 100%, which the fee manager of the module could set. However, a blocked withdrawal has no economic impact, since every exit would pay out nothing.

Acknowledged

Dromos Labs acknowledged the issue.

#025 Removing Principal Token Voting Power Can Be Undone by Anyone

Informational Version 1

Acknowledged 

In `RelayTokenVotes`, the post-transfer hook self-delegates any recipient whose delegate is unset, so delegating to the zero address to give up voting power is undone by the next deposit the holder receives. Anyone can trigger one through `requestDeposit()`, at a cost of at least `minDeposit` that the holder keeps.

Acknowledged

Dromos Labs acknowledged the issue and stated:

If the holder does not vote, the self-delegation has no effect on any proposal. The finding matters only for a contract holder that votes automatically and relies on `delegate(address(0))` to disable that vote. We treat this as an integration issue, not a bug on the token.

#027 Several State Changes Emit No Event

Informational Version 1

Code Partially Corrected / Acknowledged 

The Relay announces most of its state changes through the events declared in `IRelay`. We provide a non-exhaustive list of missing event emissions:

1. `ProtocolRelay.revokeEntrypoint()` clears a pending entrypoint proposal. The vetoer's cancellation through `vetoEntrypoint()` clears the same slot and emits `EntrypointVetoed`, so a proposal can be canceled in two ways and only one of them is visible.
2. `RelayConfigLib.clearLeafRecipient()` cancels a pending leaf recipient proposal. It emits `LeafRecipientSet` only when an active recipient was also removed, and that event reports the recipient going away rather than the proposal being withdrawn. The operator flow announces the equivalent cancellation with `OperatorProposalCancelled`.
3. `RelayConfigLib.revokeOperator()` cancels a pending operator proposal alongside clearing the configured operator. It emits `OperatorDispatched` for the dispatch, which fires whether or not a proposal was pending. The event `OperatorProposalCancelled` is reached only through `cancelOperator()`.
4. `ProtocolRelay.sweep()` sends caller-chosen token amounts to caller-chosen recipients. The equivalent sweep on the leaf side emits `TokenSwept` and `NativeSwept` for the same operation.
5. `RelayBase.emergencyDeallocate()` pulls voting weight back from a suspended chain. The sibling `evacuate()` emits `Evacuated` for the same kind of operation.
6. `RelayBase.pull()` hands unaccounted reward tokens to an entrypoint. Every other path that moves value out of the Relay announces it, through `Compounded`, `RewardNotified`, `RewardClaimed` or `WithdrawProcessed`.
7. The entrypoint contracts announce nothing for a swap, so the router, input token, input amount and minimum output the `KEEPER` chose are recorded nowhere. The configuration surface of those same contracts does emit, through `TargetTokenAdded`, `ExcludedTokenRemoved` and `CompoundWeightSet`.

Code Partially Corrected

Items 2 to 7 were fixed. Item 1 is still open: `ProtocolRelay.revokeEntrypoint()` still emits no event.

Acknowledged

Dromos Labs acknowledged the remaining item.

#028 Tokens Left on an Entrypoint Can Be Swept by Anyone

Informational Version 1

Code Corrected ✓

In `BaseEntrypoint`, `_pullSwapAndValidate()` refunds its entire `tokenIn` balance to the Relay instead of only the unspent share of `amountIn`, and the relay-agnostic entrypoints accept any `_params.relay` that reports the caller as `KEEPER`. Any token stranded on the entrypoint is therefore sweepable by anyone.

Code Corrected

Every entrypoint is now bound at construction to a single Relay through the immutable `RELAY`, and the caller-supplied `relay` parameter was removed.

#029 Undocumented Remapping of the Withdrawal Recipient

Informational Version 1

Specification Changed ✓

In `IRelay`, both `processWithdrawals()` and `withdrawals()` present the registered destination ID as fixed:

```
/// @notice Drains up to `_maxEntries` queued exits against the free chain0 weight, read
///         live from
///         the Voter. Permissionless: exits only ever route to their registered
///         destinations.
function processWithdrawals(uint256 _maxEntries) external;

/// @return _destination Destination sAERO, or `MINT_SENTINEL` when the exit mints a
///         fresh one.
function withdrawals(uint256 _id) external view returns (...);
```

However, when processing the withdrawal, `QueueLib._routeWithdrawal()` re-routes it to a freshly minted position whenever moving the weight would fail, as detected by `_destinationFallsShort()`. This occurs when the destination token does not exist, or when the destination unlocks earlier than the Relay's own position. This behavior is undocumented in the NatSpec of both functions and could lead to loss of funds if integrations are not aware of it and the recipient contract is, for example, not able to handle an incoming NFT transfer.

Specification Changed

The NatSpec of `IRelay.processWithdrawals()` now documents when an exit is paid into a freshly minted sAERO owned by the holder instead of the registered destination, namely when the destination does not exist or unlocks before the Relay's own position. It also states that the mint skips the ERC-721 receiver check. The NatSpec of `withdrawals()` and `registerOnWithdrawQueue()` refers to this rule.

#030 Identical Backing Getters Are Documented Differently

Informational Version 1

Code Corrected ✓

In `RelayBase`, `assetsBacking()` returns the public `totalBacking` state variable unchanged, so both getters always return the same value. `IRelay` documents them as if they differed:

```
/// @notice TOKEN-denominated weight backing the current shares, priced against the PT
supply.
/// @return _backing The backing counter (see `totalBacking` for the donation-inflation
guard).
function assetsBacking() external view returns (uint256 _backing);

/// @notice TOKEN-denominated weight backing the current shares: admitted deposits plus
compounds
///      minus withdrawals. An internal counter, never read from `balanceOf` or
`VE.staked()`,
///      so a donation cannot inflate the share price.
function totalBacking() external view returns (uint256 _backing);
```

An integrator cannot tell from the documentation that the two are interchangeable, and `assetsBacking()` adds nothing over `totalBacking()`.

Code Corrected

The redundant `assetsBacking()` getter was removed from `RelayBase` and `IRelay`.

#033 RelayLeafModule Cannot Bridge Partial Amounts

Informational Version 3

Acknowledged ⊖

`RelayLeafModule.bridge()` always bridges the whole `OUT_TOKEN` balance of the module. The `KEEPER` cannot choose a smaller amount.

Warp routes may cap the amount of a single transfer. For example, Hyperlane's `RateLimitedHook` reverts every dispatch above the current level of its refilling bucket, which never exceeds a maximum capacity. Its counterpart on the destination, `RateLimitedIsM`, rejects the delivery instead. A message above its maximum capacity can never be delivered. Circle's CCTP, which Hyperlane offers as a warp route for USDC, caps the amount burned per message.

If the balance of the module ever exceeds such a limit, for example due to delays in the process, large rewards, low limits or contract upgrades, every `bridge()` reverts. The `KEEPER` cannot reduce the balance, as `sweepToken()` refuses `OUT_TOKEN`. The owner can always move the whole balance out through `rescueOutToken()`, but then has to bridge the funds to the Relay manually.

Acknowledged

Dromos Labs is aware of the issue but decided not to change the code, as the owner can always recover the balance through `rescueOutToken()`.

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.