

Sablier: EVM Monorepo

Security Review

Cantina Managed review by:
0xRajeev, Lead Security Researcher
Arno, Associate Security Researcher

March 18, 2026

Contents

1 Introduction **2**
1.1 About Cantina . . . 2
1.2 Disclaimer . . . 2
1.3 Risk assessment . . . 2
1.3.1 Severity Classification . . . 2

2 Security Review Summary **3**
2.1 Scope . . . 3

3 Findings **5**
3.1 Medium Risk . . . 5
3.1.1 Redistribution rewards can lead to protocol insolvency when `AGGREGATE_AMOUNT` is misconfigured . . . 5
3.1.2 Partial Withdrawals from LPG Streams Can Permanently Brick Stream State . . . 6
3.2 Low Risk . . . 6
3.2.1 Claim consumed without token transfer if `MerkleExecute` target has fallback function and incorrect selector . . . 6
3.2.2 Sender can front-run oracle updates to cancel price-gated streams . . . 7
3.2.3 Arbitrary `selectorArguments` in `MerkleExecute` claims allow claimants to control external call behavior . . . 7
3.2.4 Sender-chosen oracle in LPG streams is risky and allows unilateral control over vested amount . . . 8
3.2.5 Absence of oracle staleness check in LPG streams can unlock or lock tokens based on outdated prices . . . 9
3.2.6 Missing L2 sequencer uptime check allows stale oracle data exploitation during sequencer downtime . . . 10
3.2.7 Attestation signatures have no expiry, preventing revocation of individual recipients . 11
3.2.8 Enabling redistribution after vesting ends creates unfair ordering-dependent reward distribution . . . 12
3.2.9 Undistributed redistribution rewards are locked in the contract when recipients never claim . . . 12
3.2.10 Model enum reordering changes integer encoding, potentially breaking offchain integrations . . . 13
3.3 Informational . . . 13
3.3.1 Missing batch support for Lockup Price-Gated streams . . . 13
3.3.2 Granularity sentinel value mismatch between airdrop campaign and Lockup stream . 14
3.3.3 Unchecked subtraction wraps to large value before timestamp validation, bypassing granularity check . . . 14
3.3.4 LPG streams missing `createWithTimestamps(...)` variant breaks API consistency and prevents precise end dates . . . 14
3.3.5 Incorrect NatSpec for LPG unlock condition specifies ”greater than” end time but code uses `>=` . . . 15
3.3.6 `MerkleExecute` does not support targets that require ETH alongside token approval 15
3.3.7 LPG oracle is not validated against the deposited token, allowing use of unrelated price feeds . . . 16
3.3.8 Enforcing 8 decimal oracles contradicts design goal of supporting sender-chosen oracles from any vendor . . . 16
3.3.9 Attested users who cannot transact are locked out due to missing delegation in attestation claims . . . 17
3.3.10 Attestor management gated by `FEE_MANAGEMENT_ROLE` violates the principle of least privilege . . . 17
3.3.11 Unused Merkle Campaign Validation Errors in Comptroller . . . 18
3.3.12 Incorrect comment for setAttestor overstates impact of setting zero address . . . 18
3.3.13 Empty array in duration-based creation functions reverts with raw panic instead of custom error . . . 19

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

Cantina Managed provides a detailed evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While Cantina Managed endeavors to identify and disclose all potential security issues, it cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities that were absent during the initial review. Therefore, any changes made to the code require a new security review to ensure that the code remains secure. Please be advised that the Cantina Managed security review is not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity level	Impact: High	Impact: Medium	Impact: Low
Likelihood: high	Critical	High	Medium
Likelihood: medium	High	Medium	Low
Likelihood: low	Medium	Low	Low

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings are a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

The Sablier Protocol is a collection of persistent, non-upgradeable smart contracts to facilitate streaming of ERC-20 tokens on Ethereum and other EVM blockchains.

From Feb 16th to Feb 24th the Cantina team conducted a review of EVM monorepo on commit hash 7c9d5d4b. The team identified a total of **25** issues:

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	0	0	0
High Risk	0	0	0
Medium Risk	2	2	0
Low Risk	10	5	5
Gas Optimizations	0	0	0
Informational	13	12	1
Total	**25**	**19**	**6**

The Cantina Managed team reviewed Sablier's EVM monorepo holistically on commit hash f9c14e2f and concluded that all findings were addressed and no new vulnerabilities were identified.

2.1 Scope

The security review had the following components in scope for EVM monorepo on commit hash 7c9d5d4b:

```
├── airdrops
│   └── src
│       ├── SablierFactoryMerkleExecute.sol
│       ├── SablierFactoryMerkleInstant.sol
│       ├── SablierFactoryMerkleLL.sol
│       ├── SablierFactoryMerkleLT.sol
│       ├── SablierFactoryMerkleVCA.sol
│       ├── SablierMerkleExecute.sol
│       ├── SablierMerkleInstant.sol
│       ├── SablierMerkleLL.sol
│       ├── SablierMerkleLT.sol
│       ├── SablierMerkleVCA.sol
│       ├── abstracts
│       │   ├── SablierFactoryMerkleBase.sol
│       │   ├── SablierMerkleBase.sol
│       │   ├── SablierMerkleLockup.sol
│       │   └── SablierMerkleSignature.sol
│       ├── libraries
│       │   ├── Errors.sol
│       │   └── SignatureHash.sol
│       └── types
│           ├── DataTypes.sol
│           ├── MerkleBase.sol
│           ├── MerkleExecute.sol
│           ├── MerkleInstant.sol
│           ├── MerkleLL.sol
│           ├── MerkleLockup.sol
│           ├── MerkleLT.sol
│           └── MerkleVCA.sol
├── flow
│   └── src
│       ├── FlowNFTDescriptor.sol
```

```
│       ├── SablierFlow.sol
│       ├── abstracts
│       │   └── SablierFlowState.sol
│       ├── libraries
│       │   ├── Errors.sol
│       │   └── FlowHelpers.sol
│       └── types
│           └── DataTypes.sol
├── lockup
│   └── src
│       ├── LockupNFTDescriptor.sol
│       ├── SablierBatchLockup.sol
│       ├── SablierLockup.sol
│       ├── abstracts
│       │   ├── SablierLockupDynamic.sol
│       │   ├── SablierLockupLinear.sol
│       │   ├── SablierLockupPriceGated.sol
│       │   ├── SablierLockupState.sol
│       │   └── SablierLockupTranched.sol
│       ├── libraries
│       │   ├── Errors.sol
│       │   ├── LockupHelpers.sol
│       │   ├── LockupMath.sol
│       │   ├── NFTSVG.sol
│       │   └── SVGElements.sol
│       └── types
│           ├── BatchLockup.sol
│           ├── DataTypes.sol
│           ├── Lockup.sol
│           ├── LockupDynamic.sol
│           ├── LockupLinear.sol
│           ├── LockupPriceGated.sol
│           └── LockupTranched.sol
└── utils
    └── src
        ├── Adminable.sol
        ├── Batch.sol
        ├── Comptrollerable.sol
        ├── NoDelegateCall.sol
        ├── RoleAdminable.sol
        ├── SablierComptroller.sol
        └── libraries
            ├── Errors.sol
            └── SafeOracle.sol
```

3 Findings

3.1 Medium Risk

3.1.1 Redistribution rewards can lead to protocol insolvency when `AGGREGATE_AMOUNT` is misconfigured

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Description: In `SablierMerkleVCA`, redistribution rewards are calculated using `AGGREGATE_AMOUNT` as a key parameter in the denominator. Because the Merkle tree is opaque on-chain, the contract cannot verify that `AGGREGATE_AMOUNT` equals the actual sum of all leaf allocations. When `AGGREGATE_AMOUNT` is accidentally set lower than the true total, the denominator in `_calculateRedistributionRewards(...)` becomes artificially small, inflating individual rewards beyond the forgone pool.

There is no cap on cumulative rewards distributed, so the contract can pay out more in redistribution rewards than was ever forgone, draining tokens meant for other recipients' base claims.

Example scenario: Four recipients in the Merkle tree: Alice (100), Bob (100), Carol (100), Dave (100). True total = 400 but AGGREGATE_AMOUNT is mistakenly set to 250. Redistribution is enabled. Vesting runs from T0 to T100.

1. Alice claims at T50: `claimAmount` = 50, `forgoneAmount` = 50. `totalForgoneAmount` = 50, `_fullAmountAllocatedToEarlyClaimers` = 100.
2. Bob claims at T50: `claimAmount` = 50, `forgoneAmount` = 50. `totalForgoneAmount` = 100, `_fullAmountAllocatedToEarlyClaimers` = 200.
3. Carol claims after T100 (vesting ended): `fullAmountAllocatedToRemainingClaimers` = 250 - 200 = 50. `rewards` = (100 * 100) / 50 = 200. `transferAmount` = 100 + 200 = 300. Carol receives 200 in rewards, which is 2x the entire forgone pool of 100.
4. Dave claims after T100: `rewards` = (100 * 100) / 50 = 200. `transferAmount` = 100 + 200 = 300. But the contract only held 400 tokens total. Alice (50) + Bob (50) + Carol (300) = 400 already spent. Dave's safeTransfer reverts and he cannot claim at all.

With the correct `AGGREGATE_AMOUNT` = 400, the denominator would be 400 - 200 = 200, giving each late claimer (100 * 100) / 200 = 50 in rewards, totaling exactly 100 and matching `totalForgoneAmount`.

Recommendation:: Consider one of the following mitigations:

1. Enforcing a cumulative reward cap: Track total rewards distributed via a `totalRedistributed` counter and enforce that they never exceed `totalForgoneAmount`. In `_postProcessClaim(...)`, after computing `rewardAmount`, cap it to the remaining pool with, for example:

```
uint128 remainingPool = totalForgoneAmount - totalRedistributed;
if (rewardAmount > remainingPool) {
    rewardAmount = remainingPool;
}
totalRedistributed += rewardAmount;
```

This prevents insolvency regardless of `AGGREGATE_AMOUNT` and allows Dave to claim his base amount without rewards in the example above. However, it does not guarantee fairness across late claimers since earlier late claimers may consume a disproportionate share of the pool before the cap kicks in.

2. Two-phase reward distribution: Separate claiming from reward collection. In phase 1, late claimers receive their `fullAmount` and their allocation is recorded. In phase 2, a separate `claimRedistributionRewards(...)` function distributes from the forgone pool using the actual tracked total of late claimer allocations as the denominator rather than the potentially incorrect `AGGREGATE_AMOUNT`. This removes the dependency on `AGGREGATE_AMOUNT` entirely for reward calculations, making the distribution correct by construction. The trade-off is requiring recipients to make a second transaction to collect redistribution rewards.

Sablier: Fixed in commit df39f78a.

Cantina Managed: Fix verified.

3.1.2 Partial Withdrawals from LPG Streams Can Permanently Brick Stream State

Severity: Medium Risk

Context: SablierLockup.sol#L313

Description: The `withdraw` function does not enforce all or nothing withdrawal semantics for LPG (Lockup Price Gated) streams, despite `calculateStreamedAmountLPG` using a binary unlock model that returns either the full `deposited` amount or `0`.

Because `withdraw` permits any `amount <= withdrawableAmount`, a recipient can withdraw a partial amount (even 1 wei) during a temporary price spike. If the oracle price subsequently drops below the target, `streamedAmount` reverts to `0` while `amounts.withdrawn` is non zero. This causes an underflow in `_withdrawableAmountOf`.

The same underflow occurs in `_cancel`.

The result is a bricked stream: both `withdraw` and `cancel` revert until either the price rises back above the target or `endTime` is reached. This can be exploited intentionally a recipient performs a 1 wei dust withdrawal during a price spike to permanently disable the sender's cancel ability at near zero cost or triggered accidentally by a legitimate partial withdrawal.

Example scenario (100 AAVE stream, target price $200):

1. Price spikes to $205 stream unlocks, `streamedAmount = 100 AAVE`.
2. Recipient withdraws 1 wei `amounts.withdrawn = 1`.
3. Price drops to $180 `streamedAmount` returns to `0`.
4. `_withdrawableAmountOf` computes `0 - 1` and underflows.
5. `_cancel` computes `recipientAmount = 0 - 1` and underflows.
6. Stream is stuck: sender cannot cancel, recipient cannot withdraw, funds are locked until the next price spike or `endTime`.

Recommendation: Enforce all or nothing withdrawal semantics for LPG streams in the `withdraw` function. If the stream model is `LOCKUP_PRICE_GATED`, require that `amount == withdrawableAmount` (i.e., the full deposited amount minus any prior withdrawals). This aligns the withdrawal behavior with the binary nature of the LPG streaming model and prevents the invariant `streamedAmount >= amounts.withdrawn` from being violated.

Sablier: Fixed in commit 12118485.

Cantina Managed: Fix verified.

3.2 Low Risk

3.2.1 Claim consumed without token transfer if `MerkleExecute` target has fallback function and incorrect selector

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: `MerkleExecute` enables an airdrop distribution model where eligible users claim tokens and immediately. execute a call on a target contract, which is useful for staking and lending pool deposits.

In `SablierMerkleExecute.claimAndExecute`, the contract approves the target, calls it with the configured selector and user-provided arguments, and then revokes the allowance. If the campaign creator configures an incorrect selector that the target does not implement and the target contract has a `fallback()` function, the low-level `TARGET.call(callData)` succeeds silently via the fallback without consuming the approved tokens. The claim bitmap is already set in `_preProcessClaim(...)` and so the recipient's claim index is permanently consumed. The tokens remain in the campaign contract but the recipient cannot claim again due to the `SablierMerkleBase_IndexClaimed` check.

Because the selector is an immutable campaign parameter, all claims in the campaign would be affected and result in every recipient losing their claim and fee without receiving tokens.

Recommendation: Consider verifying that the campaign's token balance decreased by the expected amount after the external call. If the target did not pull the tokens, revert to prevent the claim from being wasted.

Sablier: Fixed in commit bcd5b693.

Cantina Managed: Fix verified.

3.2.2 Sender can front-run oracle updates to cancel price-gated streams

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: Lockup Price-Gated (LPG) streams are an all-or-nothing vesting model where the recipient can withdraw the full deposited amount only when the oracle price reaches or exceeds a target price, or when the stream's end time is reached. Until either condition is met, the streamed amount is zero.

With `cancelable=true`, the sender can monitor the mempool for oracle price update transactions. When the sender observes an update that would push the price to or above the target price (unlocking the full deposit for the recipient), they can front-run the oracle update with a cancel call. Since `_streamedAmountOf(...)` returns 0 when the oracle price is below the target, the cancel refunds the entire deposited amount to the sender, denying the recipient their payout. This can be repeated each time the price approaches the target, effectively allowing the sender to never pay out.

Recommendation: Consider one or both of the following mitigations:

1. Add a cancel delay/timelock for LPG streams so that a cancel request takes effect after N blocks, giving the oracle time to update and the recipient time to withdraw if the price crosses the target during the delay.
2. Allow LPG stream senders to cancel only within an initial grace period (e.g., shortly after creation) to recover from configuration mistakes, after which the stream becomes non-cancelable.

Sablier: We do acknowledge the finding however have decided to not take any action at this moment partially because of contract size issue that we have in Lockup. And partially because an LPG stream requires some degree of trust between the sender and the recipient. If the sender wants to discontinue the stream, they can cancel it without needing to front run the oracle updates. Therefore, the event outlined in the finding is highly improbable.

Cantina Managed: Acknowledged.

3.2.3 Arbitrary `selectorArguments` in `MerkleExecute` claims allow claimants to control external call behavior

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: `SablierMerkleExecute` is designed to enable claim-and-execute campaigns where claimed tokens are automatically deployed into DeFi protocols such as staking or lending platforms. In `claimAndExecute(...)`, the `selectorArguments` parameter is entirely user-supplied and not included in the Merkle proof verification. While the claim index, recipient, and amount are validated against the Merkle tree, the arguments passed to the target contract are under full control of the claimant (or the UI constructing the transaction). The contract concatenates the immutable `SELECTOR` with the arbitrary `selectorArguments` and executes a low-level call on the target:

```
bytes memory callData = abi.encodePacked(SELECTOR, selectorArguments);
(bool success, bytes memory returnData) = TARGET.call(callData);
```

The `selectorArguments` parameter being entirely user-supplied and unverified is a risk to `TARGET`/campaign from the claimer. Specific risks depend on what exactly the target does with claimer funds in `TARGET.SELECTOR` and its logic. Some hypothetical examples include:

1. Lock period bypass: Claimers set lock duration to zero/minimum, immediately unstake and sell → Campaign creator's token-locking incentive alignment is completely defeated.

2. Pool/strategy redirection: Claimers route tokens to a different pool, vault, or strategy than intended → Undermines TVL goals, liquidity bootstrapping or protocol-specific staking the campaign was designed to achieve.
3. Beneficiary redirection (phishing): Attacker constructs a transaction with their own address as the staking beneficiary, tricks the legitimate claimer into signing it → Attacker receives the staked position.
4. Beneficiary redirection (intentional): Claimer deliberately stakes on behalf of a different address they control (e.g., a contract that can immediately unstake) → Used to circumvent restrictions that the staking contract applies per-address.
5. Callback/data injection: If `TARGET` function accepts a bytes data parameter forwarded to hooks or downstream contracts then claimers control arbitrary callback data → Could trigger unintended code paths, reentrancy, or manipulate internal accounting.
6. Extreme parameter values: Claimers pass boundary values (zero, `type(uint256).max`, etc.) for numerical parameters → Could exploit edge cases in the staking contract's math, create positions that are hard to liquidate/unwind, or cause overflow/underflow in poorly written targets.

Recommendation: Consider:

1. Including the expected `selectorArguments` (or a hash of them) in the Merkle leaf so that the proof verification ensures each claimant can only use the arguments specified by the campaign creator. This binds the external call parameters to the Merkle tree, removing claimant control over the target interaction.
2. Validating critical fields within `selectorArguments` at known ABI offsets using a campaign-provided hook function.
3. Validating it in the UI if onchain verification is not practical.

Sablier: We decided not to make any changes because we will validate it in the UI, and adding it would complicate the integration for others.

Cantina Managed: Acknowledged.

3.2.4 Sender-chosen oracle in LPG streams is risky and allows unilateral control over vested amount

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: The LPG model introduces an oracle-dependent vesting calculation where `calculateStreamedAmountLPG(...)` returns either 0 (nothing vested) or the full deposit (everything vested) based on the oracle price. The oracle address is chosen entirely by the sender at creation time. While `validateOracle(...)` checks that the oracle implements `AggregatorV3Interface`, returns 8 decimals and reports a positive price, it does not verify that the oracle is trustworthy, immutable or reports the price of the deposited token.

The protocol's `SECURITY.md` acknowledges this trust assumption:

> *In case of price-gated streams, there should be a certain level of trust between the sender and the recipient regarding the oracle address being used. For example, when a price-gated stream is created, it is not checked if the oracle returns the price for the deposited token or some other token.*

However, the degree of trust required is significantly understated. The risk extends beyond oracle relevance to full sender control over vesting outcomes.

A malicious sender can deploy a custom contract that passes all validation checks but has an owner-controlled `setPrice(...)` function. This gives the sender the ability to:

1. Cancel and reclaim 100% of deposited tokens at any time by setting the oracle price to 0 or negative, causing `safeOraclePrice(...)` to return 0, which makes `streamedAmount = 0` and `refundableAmount = deposited`. This is a loss-of-funds vector.

2. Prevent early unlock indefinitely by keeping the oracle price below the target, forcing the recipient to wait until `endTime`. This lack of access to funds can have significant financial implications if the end time is far in the future (e.g., months or years), as the recipient's capital is effectively locked with no recourse while the sender retains the option to cancel at any point. This is a DoS/griefing vector.

This fundamentally differs from other Lockup models (LL, LD, LT) where the streamed amount is deterministic based on elapsed time and cannot be manipulated by the sender after creation. The LPG recipient bears all risk of the sender's oracle choice.

Recommendation: Consider the following mitigations:

1. Make LPG streams non-cancelable by default or enforce non-cancelability after a short grace period, since a cancelable LPG stream with a sender-controlled oracle allows the sender to reclaim 100% of funds at any time.
2. If the protocol does not want to maintain an oracle allowlist, strengthen the trust assumption in `SECURITY.md` to explicitly warn that a malicious sender can use a custom oracle to cancel and reclaim the full deposit, not just that the oracle may report the price of a different token.
3. Surface the oracle address prominently in the UI so that recipients can independently verify the oracle before accepting an LPG stream.

Sablier: Fixed in commit 4cd8705e. As a result of the finding, we have added relevant notes in the interface of `ISablierLockupPriceGated` as well as the `SECURITY.md` file.

Cantina Managed: Fixed as recommended in (2).

3.2.5 Absence of oracle staleness check in LPG streams can unlock or lock tokens based on outdated prices

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: `SafeOracle.safeOraclePrice(...)` does not perform any staleness check on the oracle's `updatedAt` timestamp, as noted in the code:

> *Because users may not always use Chainlink oracles, we do not check for the staleness of the price.*

The rationale is that non-Chainlink oracles (e.g., Pyth, Redstone or custom oracles behind an `AggregatorV3Interface` adapter) may have different update frequencies, making a universal staleness threshold inappropriate.

However, the complete absence of staleness protection causes two issues in LPG streams:

1. Stale price above target unlocks all tokens prematurely: If the oracle last reported a price above the target but has since gone stale, `calculateStreamedAmountLPG(...)` returns the full deposit as vested. The recipient can withdraw all tokens even if the real price has since dropped below the target.
2. Stale price below target locks tokens incorrectly: If the oracle last reported a price below the target but has since gone stale, `calculateStreamedAmountLPG(...)` returns 0. The recipient's tokens remain locked even if the real price has since exceeded the target, denying them access to funds they should be entitled to.

This is notably inconsistent with `_convertUSDFeeToWei(...)` in `SablierComptroller`, which enforces a 24-hour staleness threshold for the same type of oracle data used in fee calculations, a lower-stakes context.

Recommendation: Consider:

1. Allowing the LPG stream creator to specify a `maxStaleness` parameter at creation time alongside the oracle address. `calculateStreamedAmountLPG(...)` would then check that `block.timestamp - updatedAt <= maxStaleness` before using the price, returning 0 if the price is stale. This avoids hardcoding a universal threshold in `SafeOracle` while giving each stream creator the flexibility to set an appropriate staleness bound for their chosen oracle's update frequency. Given

that the oracle is already a trusted input chosen by the sender, oracle staleness is part of that broader trust risk. However, a configurable staleness bound would provide a concrete, enforceable guardrail within the otherwise trust-dependent oracle model.

2. Adding a mechanism to allow the recipient (or both parties via mutual agreement) to force-settle the stream after a prolonged period of oracle inactivity to mitigate the issue of permanent oracle failure.

Sablier: While we agree it's a best practice to check for price staleness (and we do have this check in the comptroller, where no fee is charged if the price hasn't been updated in 24 hours), adding a staleness check to the LPG streamed amount calculation doesn't benefit the recipient because of the following scenario:

> If the target price is reached (and remains above the target) and the oracle becomes stale, users can't withdraw funds until the end time.

Cantina Managed: Acknowledged.

3.2.6 Missing L2 sequencer uptime check allows stale oracle data exploitation during sequencer downtime

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: For L2 deployments, `SafeOracle` does not check if the L2 sequencer is up or down, as recommended by Chainlink's L2 sequencer uptime feed documentation. A staleness check using `updatedAt` alone is not sufficient on L2s and does not replace the need for a sequencer uptime check. When the sequencer is down:

1. The price feed on L2 stops updating and the updatedAt timestamp freezes.
2. Sophisticated users can still submit transactions via L1.
3. These L1-submitted transactions execute on L2 before the sequencer's backlog when it comes back up.

Without a sequencer uptime check, the protocol may use stale oracle data from L1-submitted transactions, leading to incorrect LPG stream claims/cancels or incorrect fee calculations in `_convertUSDFeeToWei(...)`. This could be actively exploited by advanced users who take advantage of any price movement that occurs during the downtime or when the sequencer has just recovered.

For example, if the oracle reported a price above the LPG target before the sequencer went down and the real price has since dropped, a recipient could submit a withdrawal via L1 to claim the full deposit using the stale above-target price. Conversely, a sender could cancel a stream using a stale below-target price even though the real price has exceeded the target.

This is not a theoretical concern. For example, Base operates with a single centralized sequencer that has had two downtime incidents so far: September 2023 (~43 minutes) and August 2025 (~33 minutes).

Recommendation: Consider integrating Chainlink's L2 sequencer uptime feed into SafeOracle. Before using the oracle price, the library should check the sequencer uptime feed and return 0 (or revert in `validateOracle(...)`) if the sequencer is down or has only recently come back up, allowing a grace period for price feeds to refresh. This check should apply to both `safeOraclePrice(...)` (used in LPG streams) and `_convertUSDFeeToWei(...)` (used in fee calculations).

Sablier: After careful review, we've decided not to implement the sequencer uptime check for the following reasons:

1. LPG Stream.

Implementing this would create asymmetric outcomes that could favor either party. For example:
If the price goes above the target and the sequencer goes down, the check would block recipients from withdrawing their funds.
Similarly, if the price falls below the target during downtime, senders would be unable to cancel.
Since senders typically can always cancel the stream, we believe that in scenarios where the price exceeds the target during sequencer downtime, it's more reasonable to prioritize the recipient's ability to withdraw.

2. Convert USD Fee Function.

While the sequencer uptime check aims to prevent fee charges during downtime, the practical impact is very limited since it's only used for fee calculations of the order of $1-2. Even in a worst-case real-world scenario, the fee variance would be negligible, resulting from a few hours of price change.

3. Limited L2 Support.

Chainlink's sequencer uptime feed is not available across all major L2s we support, including Optimism and Linea. Implementing a partial solution would create inconsistent behavior across our deployments.
Plus, it will lead to maintaining a separate list of contracts because the price feed contract does not have any function for this. And we do not find it worth the impact it would have on the protocol.

Cantina Managed: Acknowledged.

3.2.7 Attestation signatures have no expiry, preventing revocation of individual recipients

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: The attestation signature verified in `SablierMerkleSignature._verifyAttestationSignature(...)` is a permanent credential with no expiry or nonce. Once the attestor signs an attestation for a recipient, it remains valid for the lifetime of the campaign. However, KYC/KYB verification, which is the primary intended use case for attestations, is inherently time-bound and revocable. A recipient who passes KYC today:

1. Could be sanctioned before the campaign ends.
2. Their KYC could expire before the campaign ends.
3. Their jurisdiction could become restricted before the campaign ends.

Yet their attestation remains valid indefinitely. The only revocation mechanism is rotating the attestor key via setAttestor, which is all-or-nothing and invalidates attestations for every unclaimed recipient in the campaign (or all campaigns using the comptroller's global attestor).

Recommendation: Consider one or more of the following mitigations:

1. Adding an expiry field to the attestation struct so that the attestor can issue time-bound attestations that must be refreshed periodically.
2. Introducing a per-recipient nonce so that individual attestations can be invalidated by incrementing the nonce without affecting other recipients.
3. Adding an on-chain blocklist maintained by the campaign admin or attestor that is checked during `claimViaAttestation(...)`, allowing targeted revocation of compromised or newly sanctioned recipients.

Sablier: Fixed in commit df39f78a.

Cantina Managed: Fix verified.

3.2.8 Enabling redistribution after vesting ends creates unfair ordering-dependent reward distribution

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: In `SablierMerkleVCA`, the admin can call `enableRedistribution(...)` at any time, including after `VESTING_END_TIME`. Recipients who already claimed their `fullAmount` before redistribution was enabled receive zero rewards, while subsequent claimers get the full proportional share of forgone tokens. This creates an ordering-dependent unfairness where identically-situated late claimers receive different payouts based solely on whether they claimed before or after the admin enabled redistribution.

The admin can also strategically time this to benefit specific addresses. For example, after observing that certain recipients have already claimed post-vesting without rewards, the admin could enable redistribution to concentrate the forgone pool among the remaining unclaimed recipients.

Recommendation: Consider restricting `enableRedistribution(...)` to `block.timestamp < VESTING_END_TIME`. This ensures redistribution is committed to before any late claimer (post-vesting) can claim. Since all late claims happen after `VESTING_END_TIME`, no late claimer can be disadvantaged by the timing of `enableRedistribution(...)`. The admin still has flexibility to watch forgone amounts accumulate during the vesting period before deciding whether to enable, but cannot time the enabling post-vesting after seeing which late claimers have already claimed without rewards.

Sablier: Fixed in commit df39f78a.

Cantina Managed: Fix verified.

3.2.9 Undistributed redistribution rewards are locked in the contract when recipients never claim

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: In `SablierMerkleVCA`, the redistribution reward denominator `AGGREGATE_AMOUNT - _fullAmountAllocatedToEarlyClaimers` includes allocations for all remaining non-early claimers, both those who eventually claim late and those who never claim at all. When recipients never claim, their share of the forgone pool is never distributed but remains locked in the contract, recoverable only by the admin via clawback within the allowed window.

Example scenario: Three recipients: Alice (100), Bob (100), Carol (100). `AGGREGATE_AMOUNT` = 300. Redistribution is enabled. Alice claims early at the midpoint, forgoing 50 tokens.

1. `totalForgoneAmount` = 50, `_fullAmountAllocatedToEarlyClaimers` = 100.
2. `fullAmountAllocatedToRemainingClaimers` = 300 - 100 = 200, because the contract assumes both Bob and Carol will claim.
3. Bob claims late: rewards = (100 * 50) / 200 = 25. Bob receives 25 out of the 50 forgone tokens.
4. Carol never claims. Her proportional share (25 tokens) remains in the contract undistributed and inaccessible outside the clawback window.

This is a capital efficiency issue where the protocol conservatively reserves Carol's share in case she claims later, but if she never does, those tokens go to nobody rather than being redistributed to Bob.

Recommendation: Consider implementing a two-phase reward distribution where claiming and reward collection are separated. In phase 1, late claimers receive their `fullAmount` and their allocation is recorded. In phase 2, a separate `claimRedistributionRewards(...)` function distributes from the forgone pool using the actual tracked total of late claimer registrations as the denominator rather than `AGGREGATE_AMOUNT`. Because the denominator is derived from actual late claimer registrations, only recipients who claim are included. In the example above, Bob would be the sole late claimer with a denominator of 100, receiving (100 * 50) / 100 = 50, which is the full forgone pool. The trade-off is requiring recipients to make a second transaction to collect redistribution rewards.

For example, the below flow ensures complete proportional fairness without any ordering bias:

1. Campaign starts (`CAMPAIGN_START_TIME`) - claims open.

2. Vesting period (`VESTING_START_TIME` → `VESTING_END_TIME`) - early claimers get partial amounts, pay fees, `totalForgoneAmount` accumulates.
3. Post-vesting (`VESTING_END_TIME` → `REWARD_EXPIRATION`) - late claimers call `claimTo(...)`, receive `fullAmount`, pay fees, their allocation is tracked.
4. Reward expiration (`REWARD_EXPIRATION`) - no more phase 1 claims. Phase 2 opens. Denominator is frozen.
5. Reward collection (`REWARD_EXPIRATION` → `EXPIRATION`) - only late claimers can call `claimRedistributionRewards(...)`, receive proportional share, pay no fee.
6. Campaign expires (`EXPIRATION`) - no more reward claims. Admin can clawback remaining balance.

The UX trade-off here is making everyone execute two transactions for fair claims+rewards versus letting them execute one transaction with potential ordering bias.

Sablier: Given our time constraints for the UI implementation, which requires following the same flow as the other campaigns, we decided not to implement it now and to consider it in future versions.

Cantina Managed: Acknowledged.

3.2.10 Model enum reordering changes integer encoding, potentially breaking offchain integrations

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: The `Lockup.Model` enum was reordered from `{LOCKUP_LINEAR=0, LOCKUP_DYNAMIC=1, LOCKUP_TRANCHED=2}` to `{LOCKUP_DYNAMIC=0, LOCKUP_LINEAR=1, LOCKUP_PRICE_GATED=2, LOCKUP_TRANCHED=3}` while adding support for LPG streams. Because Solidity stores enums as integers, this changes the ABI encoding in events, return values and storage. Any existing offchain tooling (subgraphs, indexers, frontends, analytics dashboards) that maps integer values to model names will misclassify streams from new deployments. For example, a new Linear stream (now encoded as 1) would be interpreted as Dynamic (previously encoded as 1).

Recommendation: Consider appending new enum values at the end rather than reordering existing ones. This preserves backward compatibility for off-chain tooling: `{LOCKUP_LINEAR=0, LOCKUP_DYNAMIC=1, LOCKUP_TRANCHED=2, LOCKUP_PRICE_GATED=3}`.

Sablier: Fixed in commit 8950c807.

Cantina Managed: Fix verified.

3.3 Informational

3.3.1 Missing batch support for Lockup Price-Gated streams

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: `SablierBatchLockup` provides batch creation methods for all existing Lockup models (Dynamic, Linear, and Tranched) but does not include corresponding methods for the newly introduced Price-Gated (LPG) model. The contract has `createWithDurationsLD`, `createWithTimestampsLD`, `createWithDurationsLL`, `createWithTimestampsLL`, `createWithDurationsLT` and `createWithTimestampsLT`, but no `createWithDurationsLPG` or `createWithTimestampsLPG`.

While LPG streams can still be created individually via `SablierLockup`, users who need to create multiple LPG streams in a single transaction (e.g., a company setting up price-gated vesting for multiple employees) must make separate calls, increasing gas costs and reducing UX consistency.

Recommendation: Consider adding `createWithDurationsLPG` and `createWithTimestampsLPG` to `SablierBatchLockup` following the same pattern used by the other models, along with the corresponding `BatchLockup.CreateWithDurationsLPG` and `BatchLockup.CreateWithTimestampsLPG` structs.

Sablier: Fixed in commit 7033273e.

Cantina Managed: Fix verified.

3.3.2 Granularity sentinel value mismatch between airdrop campaign and Lockup stream

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: When creating a MerkleLL campaign, a `VESTING_GRANULARITY` of 0 is stored as an immutable in `SablierMerkleLL`. However, when the claim creates the actual Lockup Linear stream, the 0 value is converted to 1 inside `SablierLockupLinear` as a sentinel (0 and 1 both mean "no discrete stepping, continuous streaming"). The on-chain streaming behavior is identical in both cases.

The issue is that offchain indexers and UIs querying the two contracts see inconsistent values: `merkleLL.VESTING_GRANULARITY()` returns 0, while `lockup.getGranularity(streamId)` returns 1 for the same logical stream. This can cause confusion in dashboards, subgraphs or analytics tools that attempt to reconcile airdrop campaign parameters with the resulting Lockup streams.

Recommendation: Normalize the granularity value at the airdrop contract level by converting 0 to 1 in the `SablierMerkleLL` constructor, so that `VESTING_GRANULARITY()` and `getGranularity(streamId)` always return the same value.

Sablier: Fixed in commit 9b81701b.

Cantina Managed: Fix verified.

3.3.3 Unchecked subtraction wraps to large value before timestamp validation, bypassing granularity check

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: In `LockupHelpers._checkTimestampsAndUnlockAmounts(...)`, when `cliffTime` is zero, the streamable range is calculated using an unchecked subtraction of `timestamps.end - timestamps.start` before the start/end time ordering is validated. If `timestamps.start > timestamps.end`, the unchecked subtraction wraps to a very large uint40 value. This wrapped `streamableRange` will likely be larger than any reasonable granularity, so the granularity check passes silently. The function eventually reverts at the `start >= end` check, but with a misleading execution path.

When `cliffTime > 0`, the ordering checks (`timestamps.start < cliffTime` and `cliffTime < timestamps.end`) happen before the unchecked subtraction, by which point `cliffTime < timestamps.end` and `timestamps.start < timestamps.end` are guaranteed, so the subtraction cannot wrap.

Recommendation: Consider moving the `timestamps.start >= timestamps.end` check into the `else` branch before the unchecked subtraction so that invalid timestamp ordering is caught first with the correct error message and without relying on a later check to catch the underflow.

Sablier: Fixed in commit 8ffd70e4.

Cantina Managed: Fix verified.

3.3.4 LPG streams missing `createWithTimestamps(...)` variant breaks API consistency and prevents precise end dates

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: All other Lockup models (Linear, Dynamic, Tranched) expose both `createWithDurations(...)` and `createWithTimestamps(...)` variants for stream creation. LPG streams only support `createWithDurationsLPG(...)` and do not have a `createWithTimestampsLPG(...)` equivalent. This was an intentional decision due to contract size constraints that prevented including both functions within the bytecode size limit.

The team chose the durations variant, but this creates a limitation: with `createWithTimestamps(...)` the duration functionality can be abstracted in the UI, but with `createWithDurations(...)` it is not possible to guarantee an exact end date since the transaction inclusion time is unknown at submission

(e.g., a user wanting the stream to end on 1st May at 00:00:00 GMT). This means LPG stream creators lack the granularity to specify precise start and end timestamps, which is particularly important for price-gated vesting where the exact end time determines when the recipient can unconditionally withdraw the full deposit regardless of oracle price.

Recommendation: Consider swapping `createWithDurationsLPG(...)` for `createWithTimestamps(...)` if only one variant can fit within the contract size limit, since the timestamps variant offers strictly greater flexibility. With `createWithTimestamps(...)`, duration-based creation can still be achieved in the UI by computing timestamps off-chain, but the reverse is not possible. If contract size allows in the future, adding both variants would bring LPG in line with other models.

Sablier: Fixed in commit ca470bc3.

Cantina Managed: Fix verified.

3.3.5 Incorrect NatSpec for LPG unlock condition specifies "greater than" end time but code uses >=

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: The `ISablierLockupPriceGated` interface documents the time-based unlock condition as:

> *The recipient can withdraw the full deposited amount when either: 1. The oracle price reaches or exceeds the target price, OR 2. Current time is greater than the stream's end time.*

However, `calculateStreamedAmountLPG(...)` in `LockupMath` uses `block.timestamp >= endTime`, which unlocks the full deposit at `endTime` itself. The NatSpec says "greater than" (implying >) while the code implements "greater than or equal to" (>=). This could mislead integrators who rely on the documented behavior to build automation around the exact unlock boundary.

Given that LL/LD/LT streams all have the `block.timestamp >= endTime` check, this is a specification error.

Recommendation: Consider aligning the NatSpec with the code by changing the documentation to:

> *Current time is greater than or equal to the stream's end time.*

or

Current time has reached the stream's end time.

Sablier: Fixed in commit 18da38f1.

Cantina Managed: Fix verified.

3.3.6 `MerkleExecute` does not support targets that require ETH alongside token approval

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: `SablierMerkleExecute.claimAndExecute(...)` approves the target to spend the claimed token amount and then executes a low-level call with the configured selector and user-provided arguments. However, the call is made with zero `msg.value`.

While `claimAndExecute(...)` is marked payable (inherited from the fee payment mechanism in `_preProcessClaim(...)`), any ETH remaining after the fee is not forwarded to the target. This means targets that require ETH alongside a token approval (e.g., protocols that charge an ETH fee for staking/depositing, or wrapping operations that need both token approval and native ETH) are not supported.

A campaign creator who sets a target expecting ETH with the call would result in all claims reverting, since the target receives no ETH. The claim fee is already consumed in `_preProcessClaim(...)`, so even if the claimant sends extra ETH, it is not forwarded to the target.

Recommendation: Consider allowing the claimant to specify an additional ETH amount to forward to the target by changing the call to `(bool success, bytes memory returnData) = TARGET.call{ value: ethAmount }(callData);`, where `ethAmount` is the remaining `msg.value` after the claim fee has been deducted. Alternatively, if this is an intentional limitation, consider documenting it in the `ISablierMerkleExecute` interface NatSpec so that campaign creators are aware that targets requiring ETH are not supported.

Sablier: Fixed in commit 18da38f1.

Cantina Managed: Fix verified.

3.3.7 LPG oracle is not validated against the deposited token, allowing use of unrelated price feeds

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: When creating an LPG stream, `validateOracle(...)` checks that the oracle implements `AggregatorV3Interface`, returns 8 decimals and reports a positive price, but does not verify that the oracle actually tracks the price of the deposited token. A sender could use an oracle for an unrelated asset accidentally, which is a risk for recipients.

This is not a theoretical concern. There have been real-world incidents where BTC/USD feeds were used for WBTC and ETH/USD feeds were used for stETH or cbETH, both assuming a 1:1 peg that caused issues during depeg events. In the LPG context, using an incorrect oracle could cause the stream to unlock prematurely (if the unrelated asset's price exceeds the target) or never unlock before `endTime` (if the unrelated asset's price remains below the target).

There is no practical on-chain mitigation since the protocol cannot trustlessly verify that an oracle corresponds to a specific token. While Chainlink feeds expose a `description()` function (e.g., "ETH / USD"), this is a string that is easily spoofed by a custom contract and does not prove the feed actually tracks that pair.

Recommendation: Consider one or more of the following mitigations:

1. Store the oracle's `description()` at creation time and surface it prominently in the UI so recipients can cross-reference the oracle's reported pair against the deposited token and identify mismatches before accepting the stream.
2. Maintain an oracle registry or allowlist at the protocol level, mapping approved oracle addresses to their corresponding tokens, so that `checkCreateLPG(...)` can validate the oracle is appropriate for the deposited token.
3. At minimum, document this risk prominently in the `ISablierLockupPriceGated` interface NatSpec and `SECURITY.md` so that recipients and integrators are aware that verifying the oracle-token relationship is their responsibility.

Sablier: Fixed in commit 4cd8705e.

Cantina Managed: Fix verified.

3.3.8 Enforcing 8 decimal oracles contradicts design goal of supporting sender-chosen oracles from any vendor

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: `SafeOracle.validateOracle(...)` enforces that the oracle returns exactly 8 decimals. While Chainlink and few others have USD price feeds use 8 decimals, there are other oracle providers that implement the `AggregatorV3Interface` use 18 decimals. Chronicle oracles, for example, return 18 decimals despite offering Chainlink-compatible adapters. API3 price feeds similarly use 18 decimals. This means LPG streams and fee calculations in `_convertUSDFeeToWei(...)` cannot use these oracle providers directly.

The LPG model is built around the premise that senders can choose their own oracle from any vendor, as evidenced by the code comment in `safeOraclePrice(...)`: *"Because users may not always use Chainlink oracles, we do not check for the staleness of the price."* However, the 8-decimal enforcement effectively

restricts oracle choice to Chainlink/Pyth/Redstone (and any others using 8-decimals), undermining this flexibility. Users who want to use Chronicle or API3 feeds must deploy wrapper contracts that convert 18-decimal prices to 8-decimal, adding complexity, gas overhead and an additional point of failure.

Recommendation: Consider making the decimal requirement configurable rather than hard-coded to 8. The oracle's `decimals()` return value could be stored alongside the oracle address at LPG stream creation time and used to normalize the price in `safeOraclePrice(...)` and `calculateStreamedAmountLPG(...)`. This would allow the protocol to support oracles of any decimal precision while maintaining correct price comparisons against the target price.

Sablier: Fixed in commit 9fb51f12.

Cantina Managed: Fix verified.

3.3.9 Attested users who cannot transact are locked out due to missing delegation in attestation claims

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: Unlike `claimViaSig(...)`, which allows anyone to submit on behalf of the recipient, `claimViaAttestation(...)` has no equivalent delegation path. The function hardcodes msg.sender as the recipient:

```
_verifyAttestationSignature(msg.sender, attestation);
_preProcessClaim({ index: index, recipient: msg.sender, ... });
```

If a recipient is attested but unable to submit transactions (e.g., recipient's newly created wallet has no native token to pay for gas, which is common with new CEX users onboarded via KYC), they cannot claim. `claimViaSig(...)` is gated to `ClaimType.DEFAULT` and is unavailable as a fallback for `ATTEST` campaigns.

Example scenario:

1. The attestor signs an attestation for Alice's wallet address.
2. Alice wants to claim but cannot submit `claimViaAttestation(...)` because her wallet has no gas.
3. Bob (a relayer/friend) offers to submit the claim on Alice's behalf.
4. Bob cannot call `claimViaAttestation(...)` because the function uses `msg.sender` as the recipient, so it would verify Bob's address against Alice's attestation and fail.
5. Bob cannot call `claimViaSig(...)` because it is gated to `ClaimType.DEFAULT` and reverts with `UnsupportedClaimType(DEFAULT, ATTEST)` for this campaign.
6. Alice is locked out until she independently funds her wallet with ETH.

Recommendation: Consider adding a `claimViaAttestationFor(...)` variant that takes an explicit recipient parameter and performs `_verifyAttestationSignature(recipient, attestation)`, similar to how `claimViaSig(...)` separates submitter from recipient. This allows any third party to submit the transaction while the attestation still binds to the intended recipient.

Sablier: We do acknowledge this but have decided to not implement it in the first version of claim-via-attestation feature. We will, however, consider it in the future upgrades.

Cantina Managed: Acknowledged.

3.3.10 Attestor management gated by `FEE_MANAGEMENT_ROLE` violates the principle of least privilege

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: In `SablierComptroller`, both `setAttestor(...)` and `setAttestorForCampaign(...)` are gated by `FEE_MANAGEMENT_ROLE`. This same role also controls fee-related functions such as `disableCustomFeeUSDFor(...)`, `setCustomFeeUSDFor(...)`, `setMinFeeUSDFor(...)` and `setMinFeeUSDForCampaign(...)`. Attestor management (controlling who can verify KYC/identity for airdrop campaigns) is a fundamentally different responsibility from fee management but yet, both are controlled by the same role.

This conflation violates the principle of least privilege. An operator granted `FEE_MANAGEMENT_ROLE` to manage protocol fees also gains the ability to rotate attestors across all campaigns, potentially invalidating pending attestations or redirecting KYC verification to a compromised key. Conversely, an operator who only needs to manage attestors must be granted fee management capabilities they do not need.

Recommendation: Consider introducing a dedicated role (e.g., `ATTESTOR_MANAGEMENT_ROLE`) for `setAttestor(...)` and `setAttestorForCampaign(...)`, separate from `FEE_MANAGEMENT_ROLE`. This allows the protocol to grant attestor management to a KYC/compliance operator without also granting them the ability to modify fees, and vice versa.

Sablier: Fixed in commit b90f531e.

Cantina Managed: Fix verified.

3.3.11 Unused Merkle Campaign Validation Errors in Comptroller

Severity: Informational

Context: SablierComptroller.sol#L252

Description: The `Errors` library defines two errors:

- `SablierComptroller_IsSablierMerkleReturnsFalse`.
- `SablierComptroller_MissingIsSablierMerkle`.

However, these errors are never used.

Recommendation: Remove the unused errors.

Sablier: Fixed in commit e4c64c0e.

Cantina Managed: Fix verified.

3.3.12 Incorrect comment for setAttestor overstates impact of setting zero address

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: In `uISablierComptroller.sol`, the comment for `setAttestor(...)` states:

> *Setting it to zero address would disable attestation-based claims globally.*

However, this is only true for campaigns that rely on the comptroller's global attestor fallback. Campaigns that have set their own local attestor via `setAttestor(...)` on the campaign contract are unaffected, since `_getAttestor(...)` in `SablierMerkleSignature` returns the local attestor when it is non-zero.

The comment is misleading and could give operators false confidence that setting the global attestor to zero disables attestation across all campaigns.

Recommendation: Consider updating the comment to accurately reflect the behavior, e.g.:

> *Setting it to zero address would disable attestation-based claims for campaigns that have not set their own local attestor.*

Sablier: Fixed in commit 8a934c0e.

Cantina Managed: Fix verified.

3.3.13 Empty array in duration-based creation functions reverts with raw panic instead of custom error

Severity: Informational

Context: SablierLockupDynamic.sol#L25

Description: When `createWithDurationsLD` is called with an empty segments array, it reverts with a raw Solidity `Panic(0x32)` from `calculateSegmentTimestamps` before reaching the proper validation. The equivalent `createWithTimestampsLD` correctly reverts with the custom error `SablierLockupHelpers_SegmentCountZero`.Same issue exists in `createWithDurationsLT`.

Recommendation: The fix is to add an early `segmentsWithDuration.length == 0` guard in `createWithDurationsLD`.

Sablier: Fixed in commit 159ff104.

Cantina Managed: Fix verified.