cecuro

Audit Report

March 1, 2026

PROJECT

Sablier

Audit Overview

Project:	Sablier
Repository:	https://github.com/sablier-labs/lockup
Audit Date:	March 1, 2026
Commit:	d0882200
Scope:	26 files

0	0	2	10	26
CRITICAL	HIGH	MEDIUM	LOW	INFO

Audit Scope

The following 26 files were included in this security audit:

- src/SablierLockup.sol
- src/LockupNFTDescriptor.sol
- src/SablierBatchLockup.sol
- src/abstracts/SablierLockupDynamic.sol
- src/abstracts/SablierLockupLinear.sol
- src/abstracts/SablierLockupState.sol
- src/abstracts/SablierLockupTranched.sol
- src/interfaces/ILockupNFTDescriptor.sol
- src/interfaces/ISablierBatchLockup.sol
- src/interfaces/ISablierLockup.sol
- src/interfaces/ISablierLockupDynamic.sol
- src/interfaces/ISablierLockupLinear.sol
- src/interfaces/ISablierLockupRecipient.sol
- src/interfaces/ISablierLockupState.sol
- src/interfaces/ISablierLockupTranched.sol
- src/libraries/Errors.sol
- src/libraries/Helpers.sol
- src/libraries/LockupMath.sol
- src/libraries/NFTSVG.sol
- src/libraries/SVGElements.sol
- src/types/BatchLockup.sol
- src/types/DataTypes.sol
- src/types/Lockup.sol
- src/types/LockupDynamic.sol
- src/types/LockupLinear.sol
- src/types/LockupTranched.sol

Findings

MEDIUM

Allowlisted recipient hooks can veto cancellations/withdrawals and permanently lock stream funds

Locations:

`src/SablierLockup.sol:364-377`

`src/SablierLockup.sol:646-659`

`src/abstracts/SablierLockupState.sol:33-35`

`src/interfaces/ISablierLockupRecipient.sol:18-60`

Description:

When a stream recipient address is allowlisted via `allowToHook`, the core contract executes recipient callbacks during withdrawals and cancellations. These callbacks are **not** wrapped in `try/catch` and the protocol **requires** that the callback returns a specific selector; otherwise it reverts.

As a result, an allowlisted recipient contract can **veto**:

- **Cancellations** (`_cancel`): hook is called unconditionally when `_allowedToHook[recipient]` is true, so any revert/invalid selector blocks the entire cancellation (including the refund to the sender).
- **Withdrawals** (`withdraw`): hook is called whenever the recipient is allowlisted and `msg.sender != recipient`, so any revert/invalid selector blocks withdrawals initiated by third parties (including approved operators and "withdraw-to-recipient" helpers).

Because allowlisting is designed to be irreversible (`_allowedToHook[recipient] = true` with no removal path), any later bug, upgrade, selfdestruct, or governance change in an allowlisted recipient contract can cause **permanent DoS** for affected streams.

This is an access-control/authorization issue because a party other than the authorized actor (the recipient's hook contract) can block actions that the protocol otherwise authorizes (sender cancellation / recipient withdrawals), potentially trapping both the sender's refundable portion and the recipient's withdrawable portion.

Evidence:

Withdrawal hook:

```
// Interaction: if `msg.sender` is not the recipient and the recipient is on the allowlist,
run the hook.
if (msg.sender != recipient && _allowedToHook[recipient]) {
 bytes4 selector = ISablierLockupRecipient(recipient).onSablierLockupWithdraw({ ... });

 // Check: the recipient's hook returned the correct selector.
 if (selector != ISablierLockupRecipient.onSablierLockupWithdraw.selector) {
 revert Errors.SablierLockup_InvalidHookSelector(recipient);
 }
}
```

(see `src/SablierLockup.sol:364-377`)

Cancellation hook:

```
// Interaction: if the recipient is on the allowlist, run the hook.
if (_allowedToHook[recipient]) {
 bytes4 selector = ISablierLockupRecipient(recipient).onSablierLockupCancel({ ... });

 // Check: the recipient's hook returned the correct selector.
 if (selector != ISablierLockupRecipient.onSablierLockupCancel.selector) {
 revert Errors.SablierLockup_InvalidHookSelector(recipient);
 }
}
```

(see `src/SablierLockup.sol:646-659`)

Impact:

- **High impact (funds stuck):** If an allowlisted recipient contract reverts (or stops returning the expected selector), senders can be prevented from canceling cancelable streams, and third parties/operators can be prevented from withdrawing on behalf of recipients. If the recipient contract cannot itself call `withdraw` (e.g., it selfdestructs or is bricked), funds can become permanently locked.
- **Operational fragility:** Any upgradeable allowlisted recipient introduces a latent "kill switch" for stream operations.

Recommendation:
Treat hooks as non-blocking notifications unless explicit veto power is intended:

- Consider wrapping hook invocations in `try/catch` (or low-level call + `success` check) and proceeding on failure.
- If veto power is intended, document it explicitly and consider mitigations (e.g., revocation mechanism, emergency bypass for sender/recipient, or code-size checks at call-time to avoid hard-locks after selfdestruct).

MEDIUM

Streams can be created with native token ERC-20 mirror if `nativeToken` is unset, risking escrow being swept as “fees”

Locations:

`src/libraries/Helpers.sol:186-222`

`src/SablierLockup.sol:293-301`

`src/interfaces/ISablierLockupState.sol:119-125`

Description:

`Helpers._checkCreateStream` attempts to block creating streams with the chain’s native token ERC-20 mirror by reverting when `token == nativeToken`.

However, `nativeToken` is initialized to `address(0)` and is set later via `SablierLockup.setNativeToken()`. While it is unset (or if the comptroller sets it to the zero address), the guard becomes ineffective for chains where the native token has a non-zero ERC-20 mirror (e.g., Polygon’s 0x…1010 described in `ISablierLockupState`).

On such chains, users could create streams using the native token’s ERC-20 mirror **before `nativeToken` is configured**. This breaks a core protocol assumption: the contract’s native balance (`address(this).balance`) can become indistinguishable from escrowed “ERC-20” stream deposits. Since `SablierLockup` inherits `Comptrollerable` (external dependency) which provides `transferFeesToComptroller()` that transfers `address(this).balance` to the comptroller, escrowed deposits for these streams can be swept out as “fees”, making withdrawals/cancellations fail.

This is a configuration-dependent but realistic footgun: the codebase does not enforce `nativeToken` being set before stream creation begins.

Evidence:

Guard relies on `nativeToken` being correctly set:

```
// Helpers.sol
// Check: the token is not the native token.
if (token == nativeToken) {
 revert Errors.SablierHelpers_CreateNativeToken(nativeToken);
}
```

`nativeToken` starts unset and can remain unset:

```
// SablierLockupState.sol
address public override nativeToken;

// SablierLockup.sol
function setNativeToken(address newNativeToken) external override onlyComptroller {
 if (nativeToken != address(0)) {
 revert Errors.SablierLockup_NativeTokenAlreadySet(nativeToken);
 }
 nativeToken = newNativeToken;
}
```

Protocol explicitly warns about native token ERC-20 mirrors:

`ISablierLockupState.nativeToken()` documents the dual-interface risk (e.g., Polygon's 0x…1010) and says these tokens cannot be used.

Impact:

* **High impact fund loss / permanent insolvency for affected streams** on chains where native balance tracks the ERC-20 mirror balance. * Escrowed "ERC-20" stream deposits may be swept to the comptroller as native fees, causing withdrawals/cancels to revert due to insufficient balance. * Even without sweeping, mixing fee payments (`msg.value`) with escrowed deposits undermines asset separation, complicating recovery and accounting.

Recommendation:

Enforce that `nativeToken` is correctly configured before allowing stream creation on chains where a native ERC-20 mirror exists (and/or disallow setting `nativeToken` to zero on such chains). Additionally, consider enforcing the `token != nativeToken` invariant at the point of stream creation even during initialization/deployment flows (e.g., via constructor/initializer configuration or an explicit "initialized" flag).

LOW

Unauthenticated `sender` lets stream creator arbitrarily select the withdrawal fee tier (fee bypass / third-party fee control)

Locations:

`src/SablierLockup.sol:663-675`

`src/abstracts/SablierLockupDynamic.sol:45-56`

`src/abstracts/SablierLockupLinear.sol:47-59`

`src/abstracts/SablierLockupTranched.sol:45-56`

`src/types/Lockup.sol:45-67`

Description:

Withdrawal fees are computed from `_streams[streamId].sender`:

```
// src/SablierLockup.sol
uint256 minFeeWei = comptroller.calculateMinFeeWeiFor({
 protocol: ISablierComptroller.Protocol.Lockup,
 user: _streams[streamId].sender
});
if (msg.value < minFeeWei) revert ...;
```

However, stream creation explicitly allows callers to provide an arbitrary `params.sender` value (documented as not necessarily equal to `msg.sender`) and stores it without any authentication/authorization from that `sender`:

```
// src/abstracts/SablierLockupDynamic.sol
sender: params.sender,

// src/SablierLockup.sol (_create)
_streams[streamId] = Lockup.Stream({ sender: sender, ... });
```

This makes the fee tier/policy effectively **user-chosen at stream creation time**, not tied to the actual funder (`msg.sender`) nor to the party that will later pay fees (recipient/withdraw caller).

Impact:

A stream creator can:

- **Bypass fee gating/revenue** by choosing a `sender` address whose `calculateMinFeeWeiFor` is low/zero.

- **Grief/lock recipients behind fees** by choosing a `sender` address that later raises its min fee (or is controlled by an adversary), making withdrawals require unexpectedly high `msg.value`.
- Render any “per-user fee policy” in the comptroller ineffective as an anti-spam or monetization mechanism, because the “user” used for fee lookups is not authenticated.

Because creation is permissionless and this choice is made at creation, likelihood is high.

Recommendation:

Bind fee policy to an authenticated identity:

- Require `params.sender == msg.sender`, **or**
- Require an authorization from `params.sender` (e.g., signature/permit-style), **or**
- Base fees on a different invariant party (e.g., `funder`/`msg.sender`/recipient) depending on intended economics.

LOW

Fees forwarding to comptroller ignores call success, can leave ETH stuck and emit misleading event

Locations:

`src/SablierLockup.sol:38-62`

`src/libraries/Errors.sol:99-101`

Description:

`SablierLockup` inherits `Comptrollerable` from `@sablier/evm-utils` which provides `transferFeesToComptroller()`. This function attempts to forward all accumulated ETH fees (`address(this).balance`) to the comptroller using a low-level `.call`, but **ignores the returned `success` flag** and emits `TransferFeesToComptroller` even if the transfer failed.

This is particularly relevant because `SablierLockup._withdraw()` enforces/collects fees via `msg.value` (so ETH can accumulate on the contract), but forwarding can silently fail if the comptroller is misconfigured (no payable `receive`/`fallback`), temporarily leaving ETH stuck in the lockup contract while offchain systems believe it was transferred.

Additionally, this repository defines `Errors.SablierLockup_FeeTransferFailed` but it is unused anywhere, suggesting the intended behavior was to revert on failed fee transfers.

Evidence:

In-scope: fee collection (ETH remains on the contract after withdrawals):

```
// src/SablierLockup.sol
uint256 minFeeWei = comptroller.calculateMinFeeWeiFor(...);
uint256 feePaid = msg.value;
if (feePaid < minFeeWei) revert ...;
// no forwarding here; ETH stays in contract balance
```

Source: `src/SablierLockup.sol:663-675`

External dependency (actual forwarding logic, `@sablier/evm-utils` v1.0.3):

```
function transferFeesToComptroller() external override {
 uint256 feeAmount = address(this).balance;
 (bool success,) = address(comptroller).call{ value: feeAmount }("");
 success; // ignored
 emit IComptrollerable.TransferFeesToComptroller(comptroller, feeAmount);
}
```

(From https://raw.githubusercontent.com/sablier-labs/evm-utils/v1.0.3/src/Comptrollerable.sol)

Unused local error suggesting intended revert-on-failure behavior:

```
// src/libraries/Errors.sol
error SablierLockup_FeeTransferFailed(address comptroller, uint256 feeAmount);
```

Impact:

- Protocol ETH fees may remain in `SablierLockup` (until comptroller is fixed/updated) without an onchain revert to signal failure.
- `TransferFeesToComptroller` event can be emitted even when no ETH moved, causing accounting/indexing discrepancies.

Recommendation:

- Check `success` and revert on failure (or bubble revert data), and consider using the existing `SablierLockup_FeeTransferFailed` error.
- Consider partial forwarding patterns or allowing comptroller to pull fees if push fails.

LOW

Irreversible setNativeToken lacks validation; misconfiguration can permanently disable streaming for a chosen ERC-20

Locations:

`src/SablierLockup.sol:293-301`

`src/abstracts/SablierLockupState.sol:25-32`

`src/libraries/Helpers.sol:186-216`

Description:

`nativeToken` is a one-time configuration set by the comptroller via `setNativeToken`. There is **no validation** on `newNativeToken` (it can be `address(0)` or any arbitrary address), yet once set it **cannot be changed**.

The `nativeToken` value is used during stream creation to reject streams whose `token` equals `nativeToken`. Therefore, setting `nativeToken` incorrectly can permanently DoS stream creation for the selected ERC-20 token address.

This is a lifecycle/configuration footgun relevant to “upgradeability” in the sense that the comptroller is expected to set protocol parameters safely over time.

Evidence:

One-time setter with no validation:

```
function setNativeToken(address newNativeToken) external override onlyComptroller {
 if (nativeToken != address(0)) {
 revert Errors.SablierLockup_NativeTokenAlreadySet(nativeToken);
 }
 nativeToken = newNativeToken;
}
```

(see `src/SablierLockup.sol:293-301`)

Creation-time check uses `nativeToken` as a banned token:

```
if (token == nativeToken) {
 revert Errors.SablierHelpers_CreateNativeToken(nativeToken);
}
```

(see `src/libraries/Helpers.sol:213-216`)

Impact:

- **Permanent DoS for a token:** If `nativeToken` is mistakenly set to an important ERC-20 address, the protocol will permanently reject creating streams for that token.
- **Irreversible misconfiguration:** Because `nativeToken` can only be set once, recovery may require redeploying the protocol contract suite.

Recommendation:

Validate `newNativeToken` before setting it (e.g., nonzero, expected wrapped native token for the chain, and/or contract-code check), or add a safer staged/confirmable configuration mechanism if irreversibility is required.

LOW

LockupNFTDescriptor abbreviateAmount uses unchecked 10**decimals with unbounded decimals (overflow/wrap ’ misleading metadata)

Locations:

`src/LockupNFTDescriptor.sol:136-171`

`src/LockupNFTDescriptor.sol:321-330`

Description:

`LockupNFTDescriptor.abbreviateAmount` computes `truncatedAmount` as `amount / 10 ** decimals` inside an `unchecked` block. The `decimals` value is taken directly from `token.decimals()` (via `safeTokenDecimals`) without bounding it to a safe range.

In Solidity >=0.8, exponentiation overflows in checked mode, but inside `unchecked` it wraps modulo 2^256. For large `decimals` (e.g., a malicious token returning 200–255), `10 ** decimals` will overflow/wrap to an essentially arbitrary 256-bit value. This can produce a misleading `truncatedAmount` (often 0, but potentially also unexpectedly large), which then affects the on-chain-rendered SVG/metadata amount.

Because Lockup can stream arbitrary ERC-20 tokens, a malicious token issuer can deliberately set/return extreme `decimals` to manipulate the displayed deposit amount.

Evidence:

```
// src/LockupNFTDescriptor.sol
unchecked {
 truncatedAmount = decimals == 0 ? amount : amount / 10 ** decimals;
}
```

`decimals` comes from:

```
// src/LockupNFTDescriptor.sol
(bool success, bytes memory returnData) = token.staticcall(abi.encodeCall(IERC20Metadata.dec-
imals, ()));
if (success && returnData.length == 32) {
 return abi.decode(returnData, (uint8));
}
```

Impact:

- Stream NFT metadata can display an incorrect deposit amount.
- This can enable phishing/deception, especially in secondary markets where buyers may rely on the displayed amount rather than validating token decimals/address.
- Funds are not directly at risk, but user decisions and off-chain integrations can be materially impacted.

Recommendation:

Constrain `decimals` to a safe maximum for `10 ** decimals` in uint256 (e.g., <= 77), and/or avoid `unchecked` exponentiation for user-controlled decimals. If out of range, fall back to a safe display path.

LOW

NFT descriptor address is not validated (zero/EOA/wrong contract), which can brick `tokenURI`/metadata

Locations:

`src/abstracts/SablierLockupState.sol:65-72`

`src/SablierLockup.sol:140-146`

`src/SablierLockup.sol:304-314`

Description:

The NFT descriptor address is taken as an unvalidated `address` in the `SablierLockupState` constructor and can later be updated via `setNFTDescriptor` without validation.

If `nftDescriptor` is set to `address(0)`, an EOA, or a contract that does not correctly implement `ILockupNFTDescriptor.tokenURI`, then `SablierLockup.tokenURI()` will revert for **all** stream NFTs. This breaks NFT metadata retrieval and can disrupt integrations (marketplaces, indexers, UIs) until the comptroller corrects the descriptor.

This is an input-validation issue because the contract accepts and stores a critical external-call target without checking it is a deployed contract or otherwise well-formed.

Evidence:

Constructor stores `initialNFTDescriptor` with no checks:

```
constructor(address initialNFTDescriptor) {
 nextStreamId = 1;
 nftDescriptor = ILockupNFTDescriptor(initialNFTDescriptor);
}
```

(SablierLockupState.sol:65-72)

`tokenURI` blindly calls into `nftDescriptor`:

```
_requireOwned({ tokenId: streamId });
uri = nftDescriptor.tokenURI({ sablier: this, streamId: streamId });
```

(SablierLockup.sol:141-146)

`setNFTDescriptor` has no validation:

```
ILockupNFTDescriptor oldNftDescriptor = nftDescriptor;
nftDescriptor = newNFTDescriptor;
```

(SablierLockup.sol:304-308)

Impact:

- Protocol-wide metadata DoS: all `tokenURI(streamId)` calls revert.
- Breaks secondary-market and UI functionality; can also break on-chain composability if other contracts depend on `tokenURI`.
- Recoverable only via comptroller calling `setNFTDescriptor` with a correct address.

Recommendation:

Validate `initialNFTDescriptor`/`newNFTDescriptor` (e.g., non-zero and `code.length > 0`) before storing, and consider additional interface/behavior checks where feasible.

LOW

Recipient hook target can become stale due to reentrancy between caching recipient and invoking hook (withdraw/cancel)

Locations:

```
src/SablierLockup.sol:338-377
src/SablierLockup.sol:625-659
src/SablierLockup.sol:637-638
src/SablierLockup.sol:701-702
```

Description:

Both `withdraw()` and `_cancel()` cache `recipient = _ownerOf(streamId)` into a local variable, then perform an **external ERC-20 transfer**, and only afterward conditionally invoke the `ISablierLockupRecipient` hook on the cached `recipient`.

Because ERC-20 transfers can be re-entrant in practice (e.g., ERC-777 style hooks, or malicious ERC-20 implementations), a re-entrant call during `token.safeTransfer(...)` can transfer the stream NFT to a new owner **before** the Lockup contract invokes the hook.

This creates a stale-recipient execution-flow bug:

- The hook may be executed on an address that is **no longer** the stream recipient at the time the hook is invoked.
- Conversely, the new recipient (current NFT owner) will **not** receive the hook.
- The `msg.sender != recipient` condition used to decide whether to hook is also evaluated against the cached (potentially stale) recipient.

Even if funds are not directly at risk (state is updated before the ERC-20 transfer), the hook mechanism is explicitly designed to let recipient contracts react to withdrawals/cancellations; reentrancy can cause callbacks to be misdirected or skipped.

Evidence:

withdraw(): cache recipient, then external transfer (inside `_withdraw`), then hook on cached recipient:

```
// src/SablierLockup.sol
address recipient = _ownerOf(streamId);
...
_withdraw(streamId, to, amount);      // includes: token.safeTransfer(to, amount)
...
if (msg.sender != recipient && _allowedToHook[recipient]) {
 ISablierLockupRecipient(recipient).onSablierLockupWithdraw(...);
}
```

And the external token transfer:

```
// src/SablierLockup.sol
token.safeTransfer({ to: to, value: amount });
```

_cancel(): cache recipient, then external refund transfer, then hook on cached recipient:

```
// src/SablierLockup.sol
address recipient = _ownerOf(streamId);
...
token.safeTransfer({ to: sender, value: senderAmount });
...
if (_allowedToHook[recipient]) {
 ISablierLockupRecipient(recipient).onSablierLockupCancel(...);
}
```

Impact:

Medium:

- **Incorrect execution flow** for integrations relying on hooks: callbacks can be delivered to the wrong contract or not delivered to the actual current recipient.
- Can lead to unexpected reverts (and thus DoS) if the cached old recipient is allowlisted and implements hook logic that now fails due to no longer owning the stream.
- Violates the interface's intent that recipient contracts can reliably "react" to these lifecycle events.

Recommendation:

- Re-fetch the current recipient (`_ownerOf(streamId)`) immediately before invoking hooks, or design hooks around immutable stream recipient rather than NFT owner.

- Alternatively, add a reentrancy guard around withdrawal/cancel paths (and/or around ERC-721 transfer paths) to prevent ownership changes during these operations.

LOW

Streams can become insolvent with fee-on-transfer or rebasing tokens (aggregateAmount and deposited amounts assume exact transfers)

Locations:

```
src/SablierLockup.sol:498-539
src/SablierLockup.sol:254-261
src/SablierLockup.sol:586-639
src/SablierLockup.sol:663-703
src/SablierBatchLockup.sol:56-88
src/SablierBatchLockup.sol:382-388
```

Description:

Lockup accounting assumes that `token.safeTransferFrom(..., depositAmount)` results in exactly `depositAmount` tokens credited to the Lockup contract.

However, for **fee-on-transfer**, **rebasing**, or otherwise non-standard ERC-20s, the actual received balance can be lower (or change over time). The protocol still records:

- `amounts.deposited = depositAmount`
- `aggregateAmount[token] += depositAmount`

This creates **unbacked liabilities**. Later withdrawals/cancellations attempt to transfer amounts derived from `deposited` even if the contract balance is insufficient, causing permanent/long-lived reverts and locking users.

Related Phase-1 suspects:

- `vp_token_compliance_SablierLockup_7c22f8dd` (verification_pass)
- `vp_token_compliance_SablierBatchLockup_5b8f87af` (verification_pass)
- `vp_token_compliance_SablierLockupLinear_7d1ee60d` (verification_pass) / rebasing
- `state-transition-and-accounting-invariant-violations` (database)

Evidence:

Accounting updated before (and without verifying) the actual received amount:

```
_streams[streamId] = Lockup.Stream({ ... amounts: Lockup.Amounts({ deposited: depositAmount,
... }) });
...
aggregateAmount[token] += depositAmount;
...
token.safeTransferFrom({ from: msg.sender, to: address(this), value: depositAmount });
```

`src/SablierLockup.sol:512-539`

Withdrawals and cancellations assume full funding:

- Withdraw transfers `amount` and decrements `aggregateAmount[token]` (-`src/SablierLockup.sol:677-703`).
- Cancel refunds `senderAmount` and decrements `aggregateAmount[token]` (-`src/SablierLockup.sol:603-639`).

`recover()` underflows when balance < aggregateAmount:

```
uint256 surplus = token.balanceOf(address(this)) - aggregateAmount[token];
```

`src/SablierLockup.sol:254-258`

Batch helper also assumes exact transfer amounts:

`SablierBatchLockup` transfers/approves the *sum* and then creates streams assuming each `depositAmount` can be pulled exactly. `src/SablierBatchLockup.sol:56-88`, `_handleTransfer`: `src/SablierBatchLockup.sol:382-388`

Concrete failure trace:

Example: 10% fee-on-transfer token. 1) Create stream with `depositAmount = 100`. 2) Lockup records `deposited=100` and `aggregateAmount += 100`, but balance increases by only `90`. 3) Recipient eventually tries to withdraw the full `100` over time. At some point, `token.safeTransfer` will revert due to insufficient balance. 4) Sender cannot recover the missing `10`; stream becomes partially/fully stuck.

Impact:

- **High-impact fund safety issue**: recipients (and/or senders on cancel) can be unable to withdraw/refund promised amounts.
- **Protocol invariants break** (`aggregateAmount` no longer matches actual backing balance), causing `recover()` to revert.

Recommendation:

Either:

- Explicitly **disallow** fee-on-transfer/rebasing tokens (document + enforce via allowlist), or
- Implement balance-before/after accounting and set `deposited` to the actual received amount (and validate tranche/segment sums accordingly), or
- Use a wrapper/token standardization approach.

Also consider guarding `recover()` against negative surplus conditions by reverting with a clearer error.

LOW

recover() lacks surplus>0 check and may revert on tokens that disallow zero-value transfers; also allows to=address(0)

Location:

`src/SablierLockup.sol:254-261`

Description:

`recover()` computes `surplus` and always calls `token.safeTransfer(to, surplus)`.

Two edge cases: 1) If `surplus == 0`, some ERC-20 tokens revert on `transfer(0)`, making `recover()` unusable. 2) There is no validation that `to != address(0)`, allowing accidental burns (comptroller-only but still a sharp edge).

This also contradicts the interface documentation which states: “The surplus amount must be greater than zero.”

Evidence:

```
uint256 surplus = token.balanceOf(address(this)) - aggregateAmount[token];
token.safeTransfer({ to: to, value: surplus });
```

`src/SablierLockup.sol:254-261`

Impact:

- Potential operational DoS for recovery on non-standard tokens.
- Potential loss of recovered funds if `to` is accidentally set to zero address.

Recommendation:

- Revert early if `surplus == 0`.
- Revert if `to == address(0)`.

LOW

LD streamed amount ‘overshoot’ fallback can freeze streaming within a segment instead of clamping

Location:

`src/libraries/LockupMath.sol:106-120`

Description:

In `LockupMath.calculateStreamedAmountLD`, if the computed streamed amount for the current segment exceeds the segment’s total amount, the function does **not** clamp to the segment amount. Instead, it returns `max(previousSegmentAmounts, withdrawnAmount)`, effectively treating the current segment’s streamed amount as **zero**.

This fallback is intended as a safety valve (“avoid locking tokens in case of a bug”), but numerically it can produce materially incorrect results (a temporary freeze in streamed amount growth until the next segment/endTime).

Evidence:

```
if (segmentStreamedAmount.gt(currentSegmentAmount)) {
 return previousSegmentAmounts > withdrawnAmount ? previousSegmentAmounts : withdrawnAmount-
;
}
```

Impact:

If this branch is ever triggered due to numerical precision issues (or unexpected behavior in underlying math), streamed amount can become artificially low, making the stream appear frozen and preventing withdrawals until later.

Recommendation:

Consider clamping to `previousSegmentAmounts + currentSegmentAmount` rather than voiding the segment, or otherwise ensuring the fallback preserves monotonicity and expected upper bounds.

LOW

SablierBatchLockup incompatible with fee-on-transfer tokens (assumes exact transfer amount received)

Locations:

`src/SablierBatchLockup.sol:56-68`

`src/SablierBatchLockup.sol:382-388`

Description:

`SablierBatchLockup` computes a `transferAmount` as the sum of `depositAmount` values and then calls:

- `token.safeTransferFrom(msg.sender, address(this), transferAmount)`
- `token.forceApprove(lockup, transferAmount)`

This assumes the batch contract receives exactly `transferAmount`. For fee-on-transfer tokens, the batch contract will receive less than `transferAmount`, and subsequent `lockup.createWith*` calls (which will attempt to `transferFrom` the batch contract for each stream's `depositAmount`) will revert due to insufficient balance.

While the entire transaction will revert (so users don't lose tokens), this is a protocol-level compatibility bug and matches the VP token compliance suspect for `SablierBatchLockup`.

Evidence:

```
// src/SablierBatchLockup.sol
for (i = 0; i < batchSize; ++i) {
 transferAmount += batch[i].depositAmount;
}
_handleTransfer(address(lockup), token, transferAmount);
...
function _handleTransfer(address lockup, IERC20 token, uint256 amount) internal {
 token.safeTransferFrom({ from: msg.sender, to: address(this), value: amount });
 _approve(lockup, token, amount);
}
```

Impact:

- Batch creation functions cannot be used with fee-on-transfer tokens (hard revert), even if the underlying Lockup contract were otherwise able to support them.

Recommendation:

- Measure actual received amount with balance-before/balance-after and/or disallow fee-on-transfer tokens explicitly.
- Alternatively, require that each `depositAmount` transfer is performed directly from the original funder to `lockup` (avoiding intermediate custody/accounting assumptions).

INFORMATIONAL

Batch-style helpers (Batch.batch and SablierBatchLockup) have unbounded iteration/return-data growth and can OOG easily at scale

Locations:

`@sablier/evm-utils/src/Batch.sol:8-32`

`src/SablierBatchLockup.sol:40-364`

Description:

The protocol exposes multiple batch-style entry points whose execution cost scales directly with user-controlled array sizes:

1) **`Batch.batch(bytes[] calls)`** (inherited by `SablierLockup` via `Batch`) iterates over `calls.length`, `delegatecall`s each payload, and stores every return blob into a `bytes[]` to return.

2) ****SablierBatchLockup.createWith*** functions iterate over `batch.length` and call into Lockup stream creation for each element. Many batch element types also include nested arrays (e.g., `segmentsWithDuration`, `segments`, `tranchesWithDuration`, `tranches`), amplifying total work and calldata size.

There are no explicit caps on these array lengths. While callers can always split into multiple transactions, these entry points are susceptible to out-of-gas and return-data/memory blowups in realistic “large airdrop / payroll” scenarios.

Evidence:

evm-utils Batch:

```
function batch(bytes[] calldata calls) external payable virtual override returns (bytes[]
memory results) {
 uint256 count = calls.length;
 results = new bytes[](count);

 for (uint256 i = 0; i < count; ++i) {
 (bool success, bytes memory result) = address(this).delegatecall(calls[i]);
 if (!success) { ... revert(...) }
 results[i] = result;
 }
}
```

SablierBatchLockup (representative):

```
uint256 batchSize = batch.length;
...
for (i = 0; i < batchSize; ++i) { transferAmount += batch[i].depositAmount; }
...
for (i = 0; i < batchSize; ++i) {
 streamIds[i] = lockup.createWithDurationsLD(..., batch[i].segmentsWithDuration);
}
```

Impact:

- **Operational scaling limitation**: large batches can revert due to block gas limits, calldata size limits, and memory growth (especially for `Batch.batch`, which returns all results).
- **Integration fragility**: integrators may assume “batch” is safe for large N, but will hit hard chain limits and unexpected failures.

Recommendation:

Add conservative batch size limits and document expected upper bounds. Consider providing pagination/chunking patterns in the periphery or off-chain tooling and avoid returning large unbounded `bytes[]` blobs where not strictly needed.

INFORMATIONAL

LockupDynamic allows exponent=0, causing immediate full segment unlock due to PRB-Math 0^0 = 1

Locations:

`src/libraries/LockupMath.sol:83-120`

`src/libraries/Helpers.sol:81-99`

`src/libraries/Helpers.sol:231-288`

`src/types/LockupDynamic.sol:8-17`

Description:

`LockupMath.calculateStreamedAmountLD` computes the per-segment streamed amount as:

- `elapsedTimePercentage = elapsedTime / segmentDuration`
- `multiplier = elapsedTimePercentage.pow(exponent)`
- `segmentStreamedAmount = multiplier * segmentAmount`

However, `Helpers.checkCreateLD` / `_checkSegments` **never validate** the `Segment.exponent` field (it only validates timestamps and amounts).

Because PRB-Math defines **`pow(0, 0) = 1e18`** (i.e., `1.0`), a segment with `exponent = 0` causes the entire segment amount to be considered streamed **immediately at the segment start** (when `elapsedTimePercentage == 0`).

This makes LD streams unexpectedly behave like a cliff/unlock-at-segment-start for exponent=0 segments, and for single-segment LD streams can make the **entire deposit immediately withdrawable at `startTime`**.

Evidence:

No exponent validation on create:

`Helpers.checkCreateLD` validates common params and segment timestamps/amount sums only:

```
// src/libraries/Helpers.sol
function checkCreateLD(..., LockupDynamic.Segment[] memory segments, ...) public pure {
 _checkCreateStream(...);
 _checkSegments(segments, depositAmount, timestamps);
}
```

`_checkSegments` checks count, ordering, end time match, and depositAmount == sum(amounts), but **never checks `segments[i].exponent`**.

Exponent used in pow at runtime:

```
// src/libraries/LockupMath.sol
SD59x18 elapsedTimePercentage = elapsedTime.div(segmentDuration);
SD59x18 multiplier = elapsedTimePercentage.pow(currentSegmentExponent);
SD59x18 segmentStreamedAmount = multiplier.mul(currentSegmentAmount);
```

At segment start, `elapsedTimePercentage == 0`.

PRB-Math semantics (root cause):

PRB-Math SD59x18 `pow` explicitly returns `UNIT` when both base and exponent are zero:

```
// @prb/math v4.1.0 src/sd59x18/Math.sol
// If both x and y are zero, the result is UNIT.
if (xInt == 0) {
 return yInt == 0 ? UNIT : ZERO;
}
```

Impact:

- **Unexpected value flow / vesting bypass for misconfigured streams**: a stream intended to vest gradually can unlock immediately at a segment boundary.
- For single-segment LD streams (common case), setting `exponent=0` makes the **full deposit withdrawable at `startTime`**, defeating time-based vesting.
- This is especially risky because the protocol enforces many LD constraints (timestamps/order/sum) but silently accepts exponent=0, which is a non-obvious footgun.

Recommendation:

Add an explicit validation rule for LD segment exponents (e.g., `exponent > 0`), or special-case the `elapsedTimePercentage == 0 && exponent == 0` case to return 0 streamed for that segment (if immediate unlock is not intended).

INFORMATIONAL

LockupNFTDescriptor unsafe ABI decoding of token metadata can revert and DoS tokenURI for some tokens

Location:

`src/LockupNFTDescriptor.sol:321-355`

Description:

`LockupNFTDescriptor.safeTokenDecimals()` and `safeTokenSymbol()` attempt to handle missing/nonstandard ERC-20 metadata using low-level `staticcall`, but they still `abi.decode` the returndata without fully validating that it is ABI-conformant for the target type.

A token can return `success = true` with malformed returndata that causes `abi.decode` to revert:

- `safeTokenDecimals`: checks `returnData.length == 32`, then `abi.decode(..., (uint8))`. If the 32-byte word does not fit into `uint8` (non-zero upper bits), decoding reverts.
- `safeTokenSymbol`: checks `returnData.length > 64`, then `abi.decode(..., (string))`. Malformed dynamic ABI (bad offset/length) reverts.

Because `tokenURI()` calls these helpers, streams using such tokens can have permanently reverting NFT metadata.

Evidence:

```
(bool success, bytes memory returnData) = token.staticcall(abi.encodeCall(IERC20Metadata.dec-
imals, ()));
if (success && returnData.length == 32) {
 return abi.decode(returnData, (uint8)); // may revert if word doesn't fit uint8
}
...
(bool success, bytes memory returnData) = token.staticcall(abi.encodeCall(IERC20Metadata.sym-
bol, ()));
if (!success || returnData.length <= 64) { return "ERC20"; }
string memory symbol = abi.decode(returnData, (string)); // may revert on malformed ABI
```

Impact:

- `tokenURI(streamId)` can revert for streams whose underlying token returns malformed but “successful” metadata responses.

- Breaks NFT metadata rendering for affected streams on marketplaces/indexers.

Recommendation:
Harden metadata decoding by validating ABI layout before decoding (e.g., check offset/length bounds for `string`, and decode `uint256` then clamp to `uint8` with range check). Alternatively, wrap `abi.decode` in `try/catch` (via external helper) or use `assembly` bounds checks to safely fall back to defaults.

INFORMATIONAL

_create() mints NFT and updates stream state before pulling ERC-20 funds, enabling ‘unfunded stream’ reentrancy during token.transferFrom

Location:

`src/SablierLockup.sol:497-539`

Description:

`_create()` writes the stream to storage, mints the recipient NFT, increments `nextStreamId`, and increments `aggregateAmount[token]` **before** calling `token.safeTransferFrom` to actually pull the deposit.

If the ERC-20 token is malicious/reentrant, its `transferFrom` can call back into `SablierLockup` while the new stream already exists and the NFT is minted, but the contract has not actually received the funds. This can enable “unfunded stream” actions during the reentrant window.

For example, if the stream `sender` is set to the token contract address, the token contract (as `msg.sender` during reentrancy) could call `cancel(streamId)` and trigger a refund transfer that pulls from the contract’s existing balance of that token (e.g., funds belonging to other streams).

This is distinct from fee-on-transfer accounting issues: it is an interaction-order semantic problem that creates a temporary inconsistent state visible to reentrant calls.

Evidence:

```
// Effects: create stream + mint NFT + bump counters
_streams[streamId] = Lockup.Stream({ ... });
_mint({ to: recipient, tokenId: streamId });
nextStreamId = streamId + 1;
aggregateAmount[token] += depositAmount;

// Interaction (happens last)
token.safeTransferFrom({ from: msg.sender, to: address(this), value: depositAmount });
```

Impact:

* With a reentrant/malicious token, an attacker can exploit the inconsistent pre-funding state to perform actions (e.g., cancel/withdraw) before funds are received. * This can lead to loss of funds held in the contract for that token (impacting other streams using the same token) or permanent accounting inconsistencies.

Recommendation:

Perform the token transfer first (or otherwise ensure no externally visible state/NFT mint occurs until after funding is secured), and/or add reentrancy protections around stream creation.

INFO

Create path lacks explicit recipient/token address validation (relies on downstream ERC721/SafeERC20 reverts)

Locations:

```
src/libraries/Helpers.sol:186-222
src/SablierLockup.sol:498-539
src/types/Lockup.sol:55-85
```

Description:

Stream creation validation (`Helpers._checkCreateStream`) does not validate some critical address inputs that are later used for external calls / token minting:

- `recipient` is never checked for `address(0)` in the helper validation.
- `token` is not checked for `address(0)` or `code.length > 0`.

Instead, creation relies on downstream failures:

- `_mint(to=recipient, ...)` reverts if `recipient == address(0)`.
- `SafeERC20.safeTransferFrom(token, ...)` reverts if `token` is not a valid ERC-20 contract (e.g., EOA/zero).

This is an input-validation gap because the contract accepts malformed addresses and only fails later with unrelated revert reasons, reducing diagnosability and making upstream validation assumptions (e.g., “parameters validated by Helpers”) incomplete.

Evidence:

Helpers validates sender/deposit/start/native-token/shape but not recipient or token-contract-ness:

```
function _checkCreateStream(
 address sender,
 uint128 depositAmount,
 uint40 startTime,
 address token,
 address nativeToken,
 string memory shape
) private pure {
 if (sender == address(0)) revert ...;
 if (depositAmount == 0) revert ...;
 if (startTime == 0) revert ...;
 if (token == nativeToken) revert ...;
 if (bytes(shape).length > 32) revert ...;
}
```

(Helpers.sol:186-222)

`_create` mints and transfers without earlier recipient/token checks:

```
_mint({ to: recipient, tokenId: streamId });
...
token.safeTransferFrom({ from: msg.sender, to: address(this), value: depositAmount });
```

(SablierLockup.sol:526-538)

Impact:

- Malformed inputs revert with low-signal errors originating from ERC721/SafeERC20 internals.
- Harder integrations and inconsistent failure modes (especially important given the protocol uses many custom errors elsewhere).

Recommendation:

Add explicit validation for `recipient != address(0)` and for `token` being a deployed contract/non-zero (as appropriate for the protocol) in the same place other create-time validations are performed.

INFO

Helpers allow arbitrary `shape` contents (only length-limited), enabling UI/log injection and phishing vectors

Locations:

`src/libraries/Helpers.sol:186-222`

`src/types/Lockup.sol:45-85`

`src/abstracts/SablierLockupDynamic.sol:137-152`

`src/abstracts/SablierLockupLinear.sol:143-159`

`src/abstracts/SablierLockupTranched.sol:137-152`

Description:

`shape` is a user-supplied string intended to "differentiate streams in the UI" (per type docs), and is emitted in stream creation events. `Helpers._checkCreateStream` only enforces `bytes(shape).length <= 32` and does not apply any charset/format restrictions.

This means a malicious stream creator can emit `shape` values containing control characters, misleading Unicode, quotes, JSON-like fragments, HTML/SVG snippets, etc. Any official or third-party UI/indexer that displays `shape` without proper escaping/sanitization can be exposed to phishing/XSS-style issues or broken rendering.

Evidence:

```
// Helpers._checkCreateStream
if (bytes(shape).length > 32) {
 revert Errors.SablierHelpers_ShapeExceeds32Bytes(bytes(shape).length);
}
```

`shape` is emitted in create events (example):

```
shape: shape
```

Impact:

* Off-chain: potential UI/log injection and phishing risk wherever `shape` is rendered unsafely. * On-chain: no direct funds risk, but it undermines the stated purpose of `shape` as a UI aid.

Recommendation:
Constrain `shape` to a safe character set (e.g., `[A-Za-z0-9 -_]`) and/or require UIs to treat it as untrusted and escape it. Consider rejecting non-UTF8 or control bytes.

INFO

ISablierLockupLinear docs reference non-existent fields (`params.unlockAmounts`, `params.timestamps.cliff`)

Locations:

`src/interfaces/ISablierLockupLinear.sol:55-92`

`src/types/Lockup.sol:65-85`

Description:

`ISablierLockupLinear.createWithTimestampsLL` documentation lists requirements involving `params.unlockAmounts` and `params.timestamps.cliff`, but `Lockup.CreateWithTimestamps` has no `unlockAmounts` field and no `cliff` timestamp field. The actual function signature passes `unlockAmounts` and `cliffTime` as separate parameters.

This is a semantic/documentation mismatch that can mislead integrators and reviewers about which values are validated.

Evidence:

Interface requirements mention:

- `params.unlockAmounts.start` / `params.unlockAmounts.cliff`
- `params.timestamps.cliff`

But the struct is:

```
// src/types/Lockup.sol
struct CreateWithTimestamps {
 address sender;
 address recipient;
 uint128 depositAmount;
 IERC20 token;
 bool cancelable;
 bool transferable;
 Timestamps timestamps;
 string shape;
}
```

And the interface function is:

```
function createWithTimestampsLL(
 Lockup.CreateWithTimestamps calldata params,
 LockupLinear.UnlockAmounts calldata unlockAmounts,
 uint40 cliffTime
) external payable returns (uint256 streamId);
```

Impact:

Off-chain callers and auditors may implement incorrect pre-validation or assume incorrect invariants (e.g., expecting `params.timestamps.cliff` to exist), leading to integration errors.

Recommendation:

Update the interface docs to refer to the actual parameters: `unlockAmounts` and `cliffTime` (and clarify that `cliffTime == 0` implies no cliff).

INFO

LockupNFTDescriptor: abbreviateAmount doc-string overstates behavior (not always prefixed with “e ”)

Location:

`src/LockupNFTDescriptor.sol:126-171`

Description:

The `abbreviateAmount()` docstring states the returned abbreviation is “prefixed with \"\u2265= \"” (e). In reality, the function returns special-case strings with different prefixes ("0", "< 1", "> 999.99T").

While this is likely intentional UX, the comment is semantically inaccurate.

Evidence:

Docstring:

```
/// @notice Creates an abbreviated representation of the provided amount, rounded down and
prefixed with ">= ".
```

Source: `src/LockupNFTDescriptor.sol:126`

Special-case returns:

```
if (amount == 0) {
 return "0";
}
...
if (truncatedAmount < 1) {
 return string.concat(SVGElements.SIGN_LT, " 1");
} else if (truncatedAmount >= 1e15) {
 return string.concat(SVGElements.SIGN_GT, " 999.99T");
}
```

Source: `src/LockupNFTDescriptor.sol:137-151`

Impact:

Documentation mismatch can mislead maintainers/integrators about formatting guarantees.

Recommendation:

Clarify the docstring to reflect the special cases (or adjust special-case outputs if the “always e” prefix is intended).

INFO

Misleading LockupMath documentation about required time bounds and linear-cliff formula

Locations:

`src/libraries/LockupMath.sol:38-43`

`src/libraries/LockupMath.sol:124-145`

Description:

Several doc comments in `LockupMath` describe stricter preconditions / different semantics than the implementation actually enforces. This can mislead integrators and future maintainers into assuming revert/overflow behavior that does not exist, or into misunderstanding the linear model's behavior around the cliff.

1) LD notes claim start/end must be in the past/future to avoid overflow/OOB:

The LD doc says:

- start time **must be in the past** "so that the calculations below do not overflow"
- end time **must be in the future** "so that the loop below does not panic with an index out of bounds error"

But the implementation explicitly handles both cases without entering the loop:

- If `startTime > block.timestamp` it returns `0`.
- If `endTime <= block.timestamp` it returns `depositedAmount`.

So the noted preconditions are not required for safety, and the comment is stale/misleading.

2) LL doc formula can be read as implying linear streaming before the cliff:

The LL doc shows a piecewise function with an `x * sa + s` branch for `block.timestamp < cliffTime`. However, the implementation returns `unlockAmounts.start` (constant) for `block.timestamp < cliffTime`:

```
if (cliffTime > blockTimestamp) {
 return unlockAmounts.start;
}
```

Whether this is “correct” depends on the intended definition of `x` and the streamable range, but as written the doc is ambiguous and can be read as implying non-zero linear streaming before the cliff, which the code does not do.

Impact:

Documentation/semantic mismatch. This can cause:

- incorrect third-party assumptions about vesting before the cliff,
- incorrect auditing/maintenance assumptions about safety requirements and failure modes.

Recommendation:

Update `LockupMath` documentation to match the actual behavior:

- clarify that start-in-future/end-in-past are handled explicitly;
- precisely define the streamable range (start’end vs cliff’end) and the intended behavior before the cliff.

INFO

Multiple create-time validations can revert with Solidity Panic on arithmetic overflow (bypassing intended custom errors)

Locations:

`src/libraries/Helpers.sol:146-184`

`src/libraries/Helpers.sol:259-287`

`src/libraries/Helpers.sol:325-354`

`src/abstracts/SablierLockupLinear.sol:36-46`

Description:

Several validation paths intended to revert with protocol custom errors can instead revert with Solidity **Panic** due to checked arithmetic overflow. This diverges from the protocol's explicit custom-error validation style and can break integrations that rely on decoding expected errors.

This is distinct from the previously reported empty-array Panic: here, the Panic can occur even after entering validation functions.

Evidence:

1) LL unlock amounts sum overflow:

In `Helpers._checkTimestampsAndUnlockAmounts`:

```
if (unlockAmounts.start + unlockAmounts.cliff > depositAmount) {
 revert Errors.SablierHelpers_UnlockAmountsSumTooHigh(...);
}
```

If `unlockAmounts.start + unlockAmounts.cliff` overflows `uint128`, the addition reverts with a Solidity Panic before the custom error can be thrown.

2) LD/LT segment/tranche sum overflow:

In `_checkSegments` and `_checkTranches`, sums are accumulated in `uint128` with checked `+=`:

```
segmentAmountsSum += segments[index].amount;
trancheAmountsSum += tranches[index].amount;
```

If the provided amounts overflow `uint128` during accumulation, the function Panics before reaching the intended custom errors (`DepositAmountNotEqualTo*Sum`).

3) LL durations overflow pre-validation:

In `createWithDurationsLL`, the contract computes `cliffTime`/`end` using `uint40` addition before calling `Helpers.checkCreateLL`:

```
cliffTime = timestamps.start + durations.cliff;
timestamps.end = timestamps.start + durations.total;
```

If either addition overflows `uint40`, the call reverts with a Panic before custom validation runs.

Impact:

- Inconsistent revert reasons for invalid inputs (Panic vs protocol custom errors).
- Tooling/integrations that expect specific custom errors for bad inputs may break.
- Harder debugging and less predictable UX.

Recommendation:

Where custom errors are intended, ensure intermediate arithmetic in validation paths cannot overflow into a Panic (e.g., widen intermediates, use checked comparisons without overflowing additions, or explicitly handle overflow cases to revert with protocol errors).

INFO

No protocol-level fee/limit on stream creation enables low-cost NFT spam and state-bloat attacks using self-minted tokens

Locations:

`src/abstracts/SablierLockupDynamic.sol:24-82`

`src/abstracts/SablierLockupLinear.sol:24-87`

`src/abstracts/SablierLockupTranched.sol:24-82`

`src/SablierLockup.sol:497-539`

`src/libraries/Helpers.sol:186-222`

Description:

Anyone can create Lockup streams (and thus mint stream NFTs) for **any recipient** as long as they provide a non-zero `depositAmount` and a token address that is not the configured `nativeToken`.

There is **no protocol fee** (other than gas) and no per-address rate limit / allowlist for stream creation. An attacker can deploy or use a self-minted ERC-20 with negligible acquisition cost, then mass-create streams with `depositAmount = 1` to:

- spam arbitrary recipients with unwanted ERC-721 NFTs,
- bloat on-chain state (`_streams`, per-model storage arrays, ERC-721 ownership),
- increase indexing/UI burdens and potentially degrade UX for targeted users.

While the attacker must pay gas, the marginal token cost can be near-zero with a mintable token, making this a viable economic griefing strategy.

Evidence:

Creation validation only enforces non-zero deposit, non-zero sender, non-zero start time, and `token != nativeToken`:

```
if (sender == address(0)) revert ...;
if (depositAmount == 0) revert ...;
if (startTime == 0) revert ...;
if (token == nativeToken) revert ...;
```

Stream creation mints an NFT to `recipient`:

```
_mint({ to: recipient, tokenId: streamId });
```

Impact:

- **Low/Medium economic griefing**: targeted NFT spam and state growth at attacker's gas expense.
- Potential operational costs for integrators/indexers and degraded UX for recipients.

Recommendation:

If unsolicited stream creation is not intended to be permissionless/spammable:

- add a creation fee (native or token),
- implement per-recipient opt-in/allowlist controls, or
- add protocol-level rate limits / minimum deposit thresholds per token or per recipient.

INFO

Percentage helpers document 4 implied decimals but implement basis-points (2 implied decimals)

Locations:

`src/LockupNFTDescriptor.sol:191-205`

`src/LockupNFTDescriptor.sol:374-385`

`src/libraries/SVGElements.sol:198-221`

Description:

`LockupNFTDescriptor.calculateStreamedPercentage()` and `stringifyPercentage()` both document their inputs/outputs as using "4 implied decimals", but the implementation (and downstream SVG rendering) clearly uses **basis points** semantics (i.e., `10_000 == 100.00%`, which is 2 implied decimals in percent units).

This is internally consistent with `SVGElements.progressCircle()` (which uses `pathLength="10000"` and `10_000 - progressNumerical`), but the documentation/comments are inconsistent and may cause incorrect assumptions in future changes or external reuse.

Evidence:

- `calculateStreamedPercentage` returns `streamedAmount * 10_000 / depositedAmount`.
- `stringifyPercentage` divides by `100` (not `10_000`) and uses `percentage % 100` as the fractional part, producing a string with **2 decimal digits**.
- `SVGElements.progressCircle` expects a 0..10_000 range.

Impact:

Primarily a correctness/documentation mismatch risk: future refactors or consumers could mis-handle the scale and display incorrect progress values.

Recommendation:

Align comments with the actual scale (basis points / 2 implied decimals in percent units), or adjust code if 4-decimal precision was intended.

INFO

Recipient hook interface does not enforce that only Lockup can call hooks (integration access-control footgun)

Location:

`src/interfaces/ISablierLockupRecipient.sol:13-60`

Description:

`ISablierLockupRecipient` defines two external hook entry points (`onSablierLockupCancel` and `onSablierLockupWithdraw`). The interface and its NatSpec do not require implementers to authenticate the caller (e.g., `require(msg.sender == address(lockup))`).

In practice, these functions are **publicly callable by any address**, with arbitrary parameter values.

This is an access-control footgun for allowlisted recipient contracts: an attacker can call the hooks directly to:

- spoof `streamId`, `sender`, `amount`, etc.
- potentially corrupt recipient-side accounting/state machines
- potentially brick legitimate later cancellations/withdrawals if the recipient hook logic becomes inconsistent

While this is primarily an implementer risk, it can surface as protocol-level DoS because allowlisted recipients execute mid-flow and their reverts propagate back to Lockup.

Evidence:

```
function onSablierLockupCancel(...) external returns (bytes4 selector);
function onSablierLockupWithdraw(...) external returns (bytes4 selector);
```

No caller authentication requirements are stated in the interface.

Impact:

- Recipient contracts that forget to restrict `msg.sender` to the Lockup contract can be attacked directly.
- Downstream effect: such attacks can cause recipient hooks to revert during real Lockup operations, creating DoS for streams that use hooks.

Recommendation:

Explicitly document (and ideally provide a reference base implementation) that recipient hooks **must** authenticate `msg.sender` as the Lockup contract (or another trusted dispatcher), and should defensively validate parameters if maintaining internal accounting.

INFO

SVG hourglass rendering depends on exact status string literals and ignores "Canceled" as terminal

Locations:

`src/libraries/SVGElements.sol:185-195`

`src/libraries/NFTSVG.sol:109-128`

Description:

`NFTSVG.generateDefs()` passes the `status` string into `SVGElements.hourglass(status)`. The hourglass rendering uses **string equality** to decide if the hourglass is “settled/depleted”:

```
bool settledOrDepleted = status.equal("Settled") || status.equal("Depleted");
...
settledOrDepleted ? "" : HOURGLASS_UPPER_BULB,
settledOrDepleted ? HOURGLASS_LOWER_BULB_LARGE : HOURGLASS_LOWER_BULB_SMALL,
```

This creates two semantic correctness risks: 1) Any change in capitalization/localization of the status text (e.g., "SETTLED", "Deplete", translated strings) will silently change the SVG rendering. 2) The cold/terminal status "Canceled" is treated the same as warm states (shows upper bulb and small lower bulb), which may be visually misleading depending on intended UX semantics.

Impact:

Low. Incorrect/misleading NFT visuals and brittle coupling between UI strings and SVG logic.

Recommendation:

Drive the hourglass state from a stable enum/flag rather than human-readable strings (e.g., pass a boolean or status enum value), and explicitly decide how "Canceled" should be rendered.

INFO

SVGElements.calculatePixelWidth is semantically brittle: miscalculates when escaped entities are not a prefix or when multiple semicolons exist

Location:

`src/libraries/SVGElements.sol:227-252`

Description:

`SVGElements.calculatePixelWidth` is named and documented as a general pixel-width estimator, but its implementation only works correctly under narrow, non-obvious constraints:

* escaped entities (e.g. `>`, `&#8805;`) must appear at the beginning of the string, and * the string must not contain any other semicolons.

If these assumptions are violated (e.g., a future caller produces text like `"1 " + SIGN_LT + " 2"` or any string containing `;` later), the function will **grossly under-estimate** the width because it subtracts `charWidth * semicolonIndex` where `semicolonIndex` is the absolute position of the last semicolon.

This is a semantic footgun: the function name suggests it can be used for arbitrary text, but the implementation is only correct for a specific formatting convention.

Evidence:

```
function calculatePixelWidth(string memory text, bool largeFont) internal pure returns (uint256
width) {
 ...
 uint256 semicolonIndex;
 for (uint256 i = 0; i < length; ++i) {
 if (bytes(text)[i] == ";") {
 semicolonIndex = i; // last semicolon wins
 }
 width += charWidth;
 }

 // Wsubtracts based on absolute index, not escaped-entity length
 width -= charWidth * semicolonIndex;
}
```

Example of incorrectness if an entity is not a prefix: * `"x " + ">"` has `semicolonIndex` near the end, causing the subtraction to remove almost the entire width.

Impact:

Incorrect width computation can lead to SVG card widths that are too small, causing text clipping/overlap in the rendered NFT image.

Recommendation:

Either: * enforce the assumptions at all call sites (e.g., ensure any escaped entity is always a prefix and never include `;` elsewhere), or * rewrite the routine to properly detect and account for HTML entities anywhere in the string (and potentially multiple entities).

INFO

SablierBatchLockup always reverts the entire batch on any per-stream failure (no partial success), which contradicts 'successfully created' expectations

Locations:

`src/SablierBatchLockup.sol:41-89`

`src/SablierBatchLockup.sol:92-148`

`src/SablierBatchLockup.sol:151-200`

`src/SablierBatchLockup.sol:203-253`

`src/SablierBatchLockup.sol:256-304`

`src/SablierBatchLockup.sol:307-363`

Description:

All `SablierBatchLockup.createWith*` functions attempt to create each stream in a simple `for` loop without `try/catch` and without any per-item error handling.

As a result:

- If any single `lockup.createWith*` call reverts, the *entire* batch transaction reverts and **no streams are created**.
- The returned `streamIds` array and the `CreateLockupBatch` event are only produced if every stream creation succeeds.

This is easy to misunderstand because the broader BatchLockup concept (and some documentation phrasing like “successfully created”) often implies partial success semantics.

Evidence:

Example (LD durations):

```
for (i = 0; i < batchSize; ++i) {
 streamIds[i] = lockup.createWithDurationsLD(...);
}

emit ISablierBatchLockup.CreateLockupBatch({ funder: msg.sender, lockup: lockup, streamIds:
streamIds });
```

Source: `src/SablierBatchLockup.sol:69-89`

No `try/catch` exists in any of the variants.

Impact:

- Integrators may incorrectly assume the batch can partially succeed and still return IDs for successes.
- Operationally, a single malformed stream spec bricks the entire batch submission.

Recommendation:

Clarify documentation/ABI expectations that batches are atomic (all-or-nothing), or implement per-item failure handling if partial success is intended.

INFO

Silent truncation of `block.timestamp` to `uint40` can break time arithmetic after year ~36812

Locations:

`src/abstracts/SablierLockupDynamic.sol:35-44`

`src/abstracts/SablierLockupLinear.sol:36-46`

`src/abstracts/SablierLockupTranched.sol:35-44`

`src/libraries/LockupMath.sol:60-70`

`src/libraries/LockupMath.sol:157-172`

`src/libraries/LockupMath.sol:240-255`

Description:

Multiple places cast `block.timestamp` (a `uint256`) down to `uint40`:

- `uint40 startTime = uint40(block.timestamp)`
- `uint40 blockTimestamp = uint40(block.timestamp)`

In Solidity, explicit narrowing casts **truncate** rather than revert. Once `block.timestamp > type(uint40).max` (~1.1e12 seconds since epoch, around year 36812), these casts wrap modulo 2^40, causing incorrect time arithmetic and therefore incorrect streamed amount calculations and stream parameter derivations.

Even though this is far in the future on typical chains, it is a real numerical boundary condition and could also manifest on non-standard chains/test environments that use unusually large timestamps.

Evidence:

```
// src/abstracts/SablierLockupDynamic.sol:36
uint40 startTime = uint40(block.timestamp);

// src/libraries/LockupMath.sol:60
uint40 blockTimestamp = uint40(block.timestamp);
```

Impact:

Informational: extreme boundary-condition bug that would break vesting schedules and streamed-amount calculations when timestamps exceed `uint40`.

Recommendation:

If long-lived safety is desired, either (a) avoid truncating casts, (b) assert that `block.timestamp <= type(uint40).max` before casting, or (c) store timestamps in a wider type.

INFO

Unchecked arithmetic on `nextStreamId` and `aggregateAmount` can silently wrap on extreme values or invariant breaks

Locations:

```
src/SablierLockup.sol:529-535
src/SablierLockup.sol:632-635
src/SablierLockup.sol:696-699
src/SablierLockup.sol:313-314
```

Description:

Several critical counters are updated inside `unchecked` blocks:

- `nextStreamId = streamId + 1`
- `aggregateAmount[token] += depositAmount`
- `aggregateAmount[token] -= senderAmount` (cancel)
- `aggregateAmount[token] -= amount` (withdraw)

While these are likely safe under assumed invariants, unchecked math means that any invariant break (e.g., unexpected token behavior, future changes, or extreme boundary conditions) can **silently wrap** and permanently corrupt accounting.

Additionally, `setNFTDescriptor()` emits `BatchMetadataUpdate(1, nextStreamId - 1)`. If `nextStreamId` ever wraps to 0 (possible after `2^256-1` stream IDs due to unchecked increment), `nextStreamId - 1` underflows and reverts, potentially breaking an admin operation.

Evidence:

```
// src/SablierLockup.sol
unchecked {
 nextStreamId = streamId + 1;
 aggregateAmount[token] += depositAmount;
}

unchecked {
 aggregateAmount[token] -= senderAmount; // cancel
}

unchecked {
 aggregateAmount[token] -= amount; // withdraw
}

emit IERC4906.BatchMetadataUpdate({ _fromTokenId: 1, _toTokenId: nextStreamId - 1 });
```

Impact:

- Silent wrap can break `recover()` arithmetic, misreport aggregate liabilities, and cause downstream logic errors.
- In extreme cases, ID wrap can cause tokenId collisions / broken metadata operations.

Recommendation:

Prefer checked arithmetic for these updates, or add explicit sanity checks (e.g., `require(nextStreamId != 0)` and `require(aggregateAmount[token] >= amount)` even if invariants are expected) to fail loudly instead of corrupting state.

INFO

`allowToHook()` contradicts its NatSpec: can revert even if already allowlisted, and lacks intended zero-code-size handling

Locations:

`src/interfaces/ISablierLockup.sol:142-156`

`src/SablierLockup.sol:163-173`

`src/libraries/Errors.sol:91-96`

Description:

`ISablierLockup.allowToHook()` states it "does not revert if the contract is already on the allowlist." However, the implementation *always* performs an external `supportsInterface` call and will revert if the recipient contract:

- has no code (EOA),
- does not implement `supportsInterface` correctly, or
- has changed/upgraded since being allowlisted.

This means a second call to `allowToHook(recipient)` can revert despite the recipient already being allowlisted, contradicting the documented behavior.

Additionally, `Errors.sol` defines `SablierLockup_AllowToHookZeroCodeSize(address)` but the implementation never uses it, and it does not pre-check `code.length`. The external call to an address with no code is likely to revert with a low-level ABI decoding error rather than a protocol-specific error.

Evidence:

NatSpec claim:

```
/// - Does not revert if the contract is already on the allowlist.
function allowToHook(address recipient) external;
```

Implementation always calls `supportsInterface` and can revert:

```
bytes4 interfaceId = type(ISablierLockupRecipient).interfaceId;
if (!ISablierLockupRecipient(recipient).supportsInterface(interfaceId)) {
 revert Errors.SablierLockup_AllowToHookUnsupportedInterface(recipient);
}
_allowedToHook[recipient] = true;
```

No early return when `_allowedToHook[recipient]` is already true, and no `recipient.code.length` check.

Unused error suggests intended behavior:

`Errors.sol` defines:

```
error SablierLockup_AllowToHookZeroCodeSize(address recipient);
```

…but it is never thrown.

Impact:

- Comptroller automation that retries allowlisting (idempotent behavior) can unexpectedly revert.
- Poor UX / opaque reverts when attempting to allowlist an EOA or non-ERC165 contract.

Recommendation:

Make allowlisting idempotent as documented (short-circuit if already allowlisted), and add explicit `code.length` handling consistent with the defined error.

INFO

`withdrawMax()` reverts when withdrawable amount is zero due to calling `withdraw()` with amount=0

Locations:

`src/SablierLockup.sol:381-384`

`src/SablierLockup.sol:347-350`

Description:

`withdrawMax()` computes `withdrawnAmount = _withdrawableAmountOf(streamId)` and unconditionally calls `withdraw(..., amount=withdrawnAmount)`.

If `withdrawableAmountOf(streamId)` is zero (common at stream start, or after full withdrawal), `withdraw()` reverts because it forbids `amount == 0`.

This is a boundary-condition mismatch: a function named `withdrawMax` often aims to be a safe "withdraw what you can" helper, but here it can unexpectedly revert instead of performing a no-op and returning 0.

Evidence:

```
function withdrawMax(uint256 streamId, address to) external payable override returns (uint128
withdrawnAmount) {
 withdrawnAmount = _withdrawableAmountOf(streamId);
 withdraw({ streamId: streamId, to: to, amount: withdrawnAmount });
}

// withdraw()
if (amount == 0) {
 revert Errors.SablierLockup_WithdrawAmountZero(streamId);
}
```

Impact:

- User-facing helper can fail at normal edge states (no tokens currently withdrawable), causing integration friction.

Recommendation:

Consider skipping the `withdraw()` call when `withdrawnAmount == 0` (similar to `withdrawMaxAndTransfer()`), or clearly document that `withdrawMax()` reverts when nothing is withdrawable.

INFO

calculateDurationInDays uses unchecked subtraction; if endTime < startTime it silently underflows and returns nonsense duration

Location:

`src/LockupNFTDescriptor.sol:173-189`

Description:

`calculateDurationInDays()` computes `(endTime - startTime) / 1 days` inside an `unchecked` block.

If `endTime < startTime` (due to upstream bug, corrupted state, or a non-Sablier contract being passed to `tokenURI()`), the subtraction underflows and yields a huge value instead of reverting, causing the function to return a misleading duration (typically the `> 9999 Days` dummy string).

This is an arithmetic boundary-condition issue: the function's name/docs imply a real duration, but invalid inputs do not fail fast.

Evidence:

`src/LockupNFTDescriptor.sol`:

```
unchecked {
 durationInDays = (endTime - startTime) / 1 days;
}
```

Impact:

- Incorrect NFT metadata display for pathological/invalid timestamp ordering.
- Makes it harder to detect upstream timestamp bugs because the underflow is masked.

Recommendation:

Remove `unchecked` or add an explicit `require(endTime >= startTime)`-style guard before performing the subtraction, so invalid inputs fail deterministically.

INFO

cancelMultiple/withdrawMultiple "does not revert on per-item failure" guarantee can be broken by large revert data (gas/memory blowup)

Locations:

`src/SablierLockup.sol:222-251`

`src/SablierLockup.sol:416-444`

`src/interfaces/ISablierLockup.sol:189-201`

Description:

`cancelMultiple()` and `withdrawMultiple()` are designed to continue even if some per-stream operations revert, emitting `InvalidStreamInCancelMultiple` / `InvalidWithdrawalInWithdrawMultiple` with the returned `bytes revertData`.

However, this “non-reverting batch” property can be defeated if an underlying revert returns **very large revert data** (e.g., from a malicious/non-standard ERC-20 token revert, or deep revert bubbling). Solidity must allocate and copy `result` into memory, and then the contract emits it in an event. Both operations have gas costs proportional to the revert-data length.

A sufficiently large revert payload can therefore:

- cause **memory expansion / out-of-gas** during `delegatecall` return-data copying
- or cause **out-of-gas** when emitting the event with large `bytes` data

Either case reverts the *entire* `cancelMultiple/withdrawMultiple` call, undoing previous successful iterations and violating the documented behavior that the function “as a whole will not revert if one or more ... revert”.

Evidence:

`cancelMultiple` logs arbitrary-length `result`:

```
(bool success, bytes memory result) = address(this).delegatecall(...);
if (!success) {
 emit ISablierLockup.InvalidStreamInCancelMultiple(streamIds[i], result);
}
```

(see `src/SablierLockup.sol:236-244`)

`withdrawMultiple` similarly logs `result` bytes:

```
(bool success, bytes memory result) = address(this).delegatecall(...);
if (!success) {
 emit ISablierLockup.InvalidWithdrawalInWithdrawMultiple(streamIds[i], result);
}
```

(see `src/SablierLockup.sol:434-442`)

Event types carry `bytes revertData`: (see `src/interfaces/ISablierLockup.sol:55-63`)

Impact:

* Batch calls can unexpectedly revert and roll back partial progress when interacting with tokens or recipients that can produce large revert payloads. * This undermines the reliability of batch operations and can be used for griefing in edge cases.

Recommendation:

Bound the size of revert data that is retained/emitted (e.g., truncate to a fixed maximum) or avoid emitting the raw revert data entirely.

INFO

setNFTDescriptor emits BatchMetadataUpdate with an invalid range when no streams exist (from=1,to=0)

Location:

`src/SablierLockup.sol:303-314`

Description:

`setNFTDescriptor` unconditionally emits `BatchMetadataUpdate({ _fromTokenId: 1, _toTokenId: nextStreamId - 1 })`. When no streams exist, `nextStreamId` is 1, so `_toTokenId` becomes 0, producing an inverted/invalid range (1..0).

Evidence:

```
emit IERC4906.BatchMetadataUpdate({ _fromTokenId: 1, _toTokenId: nextStreamId - 1 });
```

With the constructor initializing `nextStreamId = 1`, `_toTokenId` is `0` before any stream is created.

Impact:

Off-chain indexers and clients may mis-handle the event or interpret it incorrectly, potentially leading to missed metadata refreshes.

Recommendation:

Guard the event emission so it is only emitted when `nextStreamId > 1`, or emit a range consistent with ERC-4906 expectations.

INFO

supportsInterface() does not advertise ISablierLockup interfaceId (ERC-165 mismatch for a contract that claims to implement ISablierLockup)

Location:

`src/SablierLockup.sol:133-137`

Description:

`SablierLockup` inherits `ISablierLockup`, which (via `IERC4906`) is ERC-165 aware. However, `supportsInterface()` only special-cases the ERC-4906 interface id and then defers to `ERC721`.

As a result, `supportsInterface(type(ISablierLockup).interfaceId)` (and other Sablier-specific interface IDs) will likely return `false` even though the contract implements those functions.

This is a semantic mismatch for integrators that rely on ERC-165 to detect the full Lockup surface.

Evidence:

```
function supportsInterface(bytes4 interfaceId) public view override(IERC165, ERC721) returns
(bool) {
 // 0x49064906 is the ERC-165 interface ID required by ERC-4906
 return interfaceId == 0x49064906 || super.supportsInterface(interfaceId);
}
```

Impact:

* On-chain/off-chain integrations that check ERC-165 for `ISablierLockup` may incorrectly conclude the contract is not a Lockup instance.

Recommendation:

Include the interface IDs for Sablier's own ERC-165 interfaces in `supportsInterface` (e.g., `type(ISablierLockup).interfaceId`), if ERC-165 discoverability is intended.

INFO

tokenURI recomputes streamed amount twice (statusOf + streamedAmountOf), risking out-of-gas metadata queries for complex schedules

Locations:

`src/LockupNFTDescriptor.sol:57-120`

`src/SablierLockup.sol:118-146`

`src/abstracts/SablierLockupState.sol:242-258`

Description:

`LockupNFTDescriptor.tokenURI()` calls `lockup.statusOf(streamId)` and `lockup.streamedAmountOf(streamId)`.

For warm streams, `statusOf()` calls `_statusOf()`, which calls `_streamedAmountOf()` to decide between `STREAMING` and `SETTLED`. The descriptor then calls `streamedAmountOf()` which calls `_streamedAmountOf()` again.

This doubles the most expensive computation in the tokenURI path for LD/LT streams (streamed amount calculation can traverse / copy segment or tranche arrays; see separate finding). As a result, tokenURI calls are much more likely to exceed typical `eth_call` gas limits used by RPC providers and indexers, effectively bricking NFT metadata retrieval for streams with larger schedules.

Evidence:

Descriptor calls both functions:

```
// src/LockupNFTDescriptor.sol
vars.status = stringifyStatus(vars.lockup.statusOf(streamId));
vars.streamedPercentage = calculateStreamedPercentage({
 streamedAmount: vars.lockup.streamedAmountOf(streamId),
 depositedAmount: vars.depositedAmount
});
```

`statusOf` delegates to `_statusOf`, which calls `_streamedAmountOf` for warm streams:

```
// src/abstracts/SablierLockupState.sol
if (_streamedAmountOf(streamId) < _streams[streamId].amounts.deposited) {
 return Lockup.Status.STREAMING;
} else {
 return Lockup.Status.SETTLED;
}
```

Impact:

- Increased likelihood that `tokenURI()` reverts or times out for LD/LT streams with larger segment/tranche arrays.
- Ecosystem impact: marketplaces/indexers fail to fetch metadata, making NFTs harder to display/trade.

Recommendation:

Avoid recomputing streamed amount in `tokenURI` by retrieving streamed amount once and deriving the status from already-fetched data when possible, or by calling a cheaper status endpoint that does not recompute streamed amount.

INFO

withdrawMaxAndTransfer emits two ERC-4906 MetadataUpdate events for the same streamId (withdraw + transfer)

Locations:

`src/SablierLockup.sol:386-414`

`src/SablierLockup.sol:316-377`

`src/SablierLockup.sol:541-562`

Description:

`withdrawMaxAndTransfer()` calls `withdraw()` (which emits `IERC4906.MetadataUpdate`) and then transfers the NFT (which triggers `_update`, which also emits `IERC4906.MetadataUpdate`).

The interface/docs for `withdrawMaxAndTransfer` imply a single metadata update, but the implementation emits **two** metadata update events for the same tokenId when a withdrawal occurs.

Evidence:

```
// withdraw emits MetadataUpdate
emit IERC4906.MetadataUpdate({ _tokenId: streamId });

// _update emits MetadataUpdate on transfer
emit IERC4906.MetadataUpdate({ _tokenId: streamId });
```

Impact:

* Off-chain indexers/UIs may perform redundant refreshes or show duplicate notifications.

Recommendation:

Avoid emitting duplicate metadata update events for the same operation path (e.g., emit once per transaction or gate one of the emissions).

Advisory to Clients

Cecuro highly recommends scheduling a follow-up security assessment within six months of this report or immediately after any significant modifications to the codebase, whichever occurs first. This practice is essential for preserving the project's security posture and identifying potential vulnerabilities that may arise from code changes or updates.

Disclaimer

The smart contracts submitted for analysis have been reviewed using Cecuro.ai's AI security analysis system, applying industry-standard vulnerability detection patterns and best practices at the time of this report. The findings disclosed in this report reflect the AI system's assessment of the submitted source code.

This report contains no statements or warranties on the identification of all vulnerabilities or the overall security of the code. Any security review methodology may not detect all potential issues, particularly novel attack vectors, complex business logic flaws, or economic exploits. The report covers the code submitted at the specified version and may not be relevant after any modifications.

Do not consider this report as a final and sufficient assessment regarding the security, utility, or bug-free status of the code. We recommend complementing this analysis with additional independent audits and a public bug bounty program to strengthen the security posture of your smart contracts.

Smart contracts are deployed and executed on blockchain platforms. The platform, its programming language, compiler, and other related software may contain vulnerabilities beyond the scope of code-level analysis. Cecuro.ai cannot guarantee the explicit security of the analyzed smart contracts.

This report does not constitute financial, legal, or investment advice. It is not a recommendation to buy, sell, or invest in any token or project, nor an endorsement of any kind. Users should conduct their own independent due diligence.

cecuro

Agentic Smart Contract Auditing