ETHGas

Token, Rebate, Escrow and Distributor

Security Assessment

Version: 3.2

January, 2026

Contents

Introduction

Sigma Prime was commercially engaged to perform a time-boxed security review of the ETHGas components in scope. The review focused solely on the security aspects of the Solidity implementation of the contract, though general recommendations and informational comments are also provided.

Disclaimer

Sigma Prime makes all effort but holds no responsibility for the findings of this security review. Sigma Prime does not provide any guarantees relating to the function of the components in scope. Sigma Prime makes no judgements on, or provides any security review, regarding the underlying business model or the individuals involved in the project.

Document Structure

The first section provides an overview of the functionality of the ETHGas components contained within the scope of the security review. A summary followed by a detailed review of the discovered vulnerabilities is then given which assigns each vulnerability a severity rating (see Vulnerability Severity Classification), an *open/closed/resolved* status and a recommendation. Additionally, findings which do not have direct security implications (but are potentially of interest) are marked as *informational*.

The appendix provides additional documentation, including the severity matrix used to classify vulnerabilities within the ETHGas components in scope.

Overview

The in-scope files consist of three core contracts responsible for the EthGas rebate and token-lock ecosystem.

- `EthgasRebate.sol` is the primary Merkle-based reward distributor that allows whitelisted or public users to claim multi-token rebates.

- `EthgasTokenLock.sol` implements a customizable linear unlock + optional revocable vesting schedule for team/advisor allocations, with support for staking released tokens, surplus withdrawal, snapshot delegation, and reward claiming.

- `VotingEscrow.vy` is a voting-escrow implementation that tracks time-weighted voting power, supports permanent locks of up to 4 years

- `FeeDistributor.vy` is a fee distribution contract that performs checkpoints, tracks supply, and lets holders claim their rewards.

Security Assessment Summary

Scope

The review was conducted on the files hosted on the ethgas-developer//ethgas-contracts-core-new-for-audit and ethgas-developer/ethgas-contracts-dao-for-audit repository.

The scope of this time-boxed review was strictly limited to files at commit:

1. 3155d40
2. ce0942b.

In addition the PRs:

1. pull/1
2. pull/1

For the following files:

1. EthgasRebate.sol
2. EthgasTokenLock.sol
3. VotingEscrow.vy
4. FeeDistributor.vy

After the review, the contracts were deployed at:

1. Voting Escrow: 0x13aB49189EBC2287E941a82D9Af154130f96Eb21
2. Rebate: 0x96C1a1a2a0cf75DF0B85661B8c99F40ABF39F6A9
3. Fee Distributor: 0x2Bd8849CCdf22B53eE10C83529B9863Ff9af2c71

Note: third party libraries and dependencies were excluded from the scope of this assessment.

Approach

The security assessment covered components written in Solidity and Vyper.

For the Solidity components, the manual review focused on identifying issues associated with the business logic implementation of the contracts. This includes their internal interactions, intended functionality and correct implementation with respect to the underlying functionality of the Ethereum Virtual Machine (for example, verifying correct storage/memory layout).

Additionally, the manual review process focused on identifying vulnerabilities related to known Solidity anti-patterns and attack vectors, such as re-entrancy, front-running, integer overflow/underflow and correct visibility specifiers.

For a more detailed, but non-exhaustive list of examined vectors, see [?, ?].

To support the Solidity components of the review, the testing team may use the following automated testing tools:

- Aderyn: `https://github.com/Cyfrin/aderyn`
- Slither: `https://github.com/trailofbits/slither`
- Mythril: `https://github.com/ConsenSys/mythril`

Output for these automated tools is available upon request.

Coverage Limitations

Due to the time-boxed nature of this review, all documented vulnerabilities reflect best effort within the allotted, limited engagement time. As such, Sigma Prime recommends to further investigate areas of the code, and any related functionality, where majority of critical and high risk vulnerabilities were identified.

Findings Summary

The testing team identified a total of 17 issues during this assessment. Categorised by their severity:

- Critical: 1 issue.
- High: 1 issue.
- Medium: 6 issues.
- Low: 5 issues.
- Informational: 4 issues.

Detailed Findings

This section provides a detailed description of the vulnerabilities identified within the ETHGas components in scope. Each vulnerability has a severity classification which is determined from the likelihood and impact of each issue by the matrix given in the Appendix: Vulnerability Severity Classification.

A number of additional properties of the components, including optimisations, are also described in this section and are labelled as “informational”.

Each vulnerability is also assigned a **status**:

- ***Open:*** the issue has not been addressed by the project team.
- ***Resolved:*** the issue was acknowledged by the project team and updates to the affected components(s) have been made to mitigate the related risk.
- ***Closed:*** the issue was acknowledged by the project team but no further actions have been taken.

Summary of Findings

ID	Description	Severity	Status
EGA-01	Lack Of Access Control On `VotingEscrow.deposit_for()`	Critical	Resolved
EGA-02	Revoked Tokens Counted As Surplus Allow Beneficiary To Bypass Revocation	High	Resolved
EGA-03	Users Can Lose Rewards By Providing Incomplete Token List	Medium	Closed
EGA-04	Beneficiary Can Grief Admin By Staking Unvested Tokens	Medium	Resolved
EGA-05	Per-Category Minimum Unlock Duration Not Enforced For Existing Locks	Medium	Resolved
EGA-06	Front-Running Revoke Allows Beneficiary To Keep Snapshot Voting Power	Medium	Resolved
EGA-07	Missing Acceptance Check Before Staking Allows Admin To Sweep Returned Funds	Medium	Resolved
EGA-08	Inconsistent Managed Amount And Period Configuration Causes Dust Amounts	Medium	Closed
EGA-09	Staking Rewards For Vested Tokens Can Be Claimed By Admin After Revocation	Low	Resolved
EGA-10	Division By Zero In Vesting View Functions When Revocable Is False	Low	Resolved
EGA-11	Integer Overflow In Unlock Time Validation	Low	Resolved
EGA-12	Missing Validation For Constructor Amounts	Low	Resolved
EGA-13	Beneficiary-Controlled Rotation Allows Pre-Change Draining Of Lock	Low	Closed
EGA-14	Missing View Functions For Contract State	Informational	Resolved
EGA-15	Unused SmartWalletChecker Interface	Informational	Resolved
EGA-16	Missing Reentrancy Guard On Deposit Functions	Informational	Resolved
EGA-17	Miscellaneous General Comments	Informational	Resolved

EGA-01	Lack Of Access Control On `VotingEscrow.deposit_for()`		
Asset	`VotingEscrow.vy, EthgasTokenLock.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

Lack of access control on `VotingEscrow.deposit_for()` allows arbitrary callers to force-lock tokens belonging to any address that has approved the escrow contract, including the `EthgasTokenLock` vault, effectively freezing all managed funds until the existing ve-lock expiry and breaking the expected vesting and release schedule.

In `VotingEscrow.vy`, the `deposit_for()` function does not enforce any whitelist or non-contract checks and unconditionally spends tokens from the `_addr` parameter, rather than from `msg.sender`:

```
@external
@nonreentrant('lock')
def deposit_for(_addr: address, _value: uint256):
    """
    @notice Deposit `_value` tokens for `_addr` and add to the lock
    @dev Anyone (even a smart contract) can deposit for someone else, but
         cannot extend their locktime and deposit for a brand new user
    @param _addr User's wallet address
    @param _value Amount to add to user's lock
    """
    _locked: LockedBalance = self.locked[_addr]

    assert _value > 0  # dev: need non-zero value
    assert _locked.amount > 0, "No existing lock found"
    assert _locked.end > block.timestamp, "Cannot add to expired lock. Withdraw"

    self._deposit_for(_addr, _addr, _value, 0, self.locked[_addr], DEPOSIT_FOR_TYPE)  # @audit _addr is also the funding address,
        ↪  with no whitelist on caller
```

The internal `_deposit_for()` implementation uses `_funding_addr` here (`_addr`) as the source of funds:

```
_locked: LockedBalance = locked_balance
supply_before: uint256 = self.supply

self.supply = supply_before + _value
old_locked: LockedBalance = _locked
# Adding to existing lock, or if a lock is expired - creating a new one
_locked.amount += convert(_value, int128)
if unlock_time != 0:
    _locked.end = unlock_time
self.locked[_addr] = _locked
...
if _value != 0:
    assert ERC20(self.token).transferFrom(_funding_addr, self, _value)  # @audit spends from `_addr`'s allowance, not from caller
```

In `EthgasTokenLock`, the constructor grants an unlimited approval to `veToken` for all managed tokens, and the `stake()` function can create or grow a ve-lock for the lock contract's own address:

```
aclManager = _aclManager;
beneficiary = _beneficiary;
token = _token;
veToken = _veToken;
feeDistributor = _feeDistributor;
delegateRegistry = _delegateRegistry;

managedAmount = _managedAmount;
...
token.approve(address(veToken), type(uint256).max); //@audit unlimited approval to VotingEscrow
```

```
function stake(uint256 _amount, uint256 _initUnlockTime) external onlyBeneficiary {
    require(_amount <= currentBalance(), "No available balance");
    IVotingEscrow.LockedBalance memory l = veToken.locked(address(this));
    if (l.amount > 0) {
        veToken.increase_amount(_amount);
    } else {
        veToken.create_lock(_amount, _initUnlockTime);
    }
    emit TokensStaked(address(this), _amount);
}
```

While `create_lock()`, `create_lock_for()`, `increase_amount()` and `increase_amount_for()` in `VotingEscrow.vy` all enforce either `assert_not_contract(msg.sender)` or `assert self.whitelisted_contracts[msg.sender]`, `deposit_for()` has no such guard. Consequently:

- Once an address `_addr` has an existing non-expired lock (`locked[_addr].amount > 0`), any externally owned account or contract can call `deposit_for(_addr, _value)` and cause `VotingEscrow` to pull `_value` tokens from `_addr` using ERC-20 `transferFrom()`, provided `_addr` has approved the escrow contract.
- For `EthgasTokenLock`, the constructor's unlimited approval combined with a single `stake()` call (which creates a long-dated lock for `address(this)`) means that an attacker can subsequently call `deposit_for(address(EthgasTokenLock), value)` to drain the entire remaining token balance from the lock contract into the ve-lock. Those tokens are then locked until the ve-lock's `end` timestamp, which may be up to four years in the future.

The same pattern applies to any end-user address that has granted `VotingEscrow` a large allowance, an attacker can force-lock the user's approved tokens by calling `deposit_for(user, value)`, causing the user's tokens to "disappear" into a ve-lock until expiry without their explicit action.

Because this behaviour allows relatively trivial, permissionless griefing of both vault beneficiaries and end-users, and can immobilise all managed funds for the full lock duration.

Recommendations

Restrict `deposit_for()` like the other function so that only explicitly authorised callers can lock tokens on behalf of another address, and avoid spending from arbitrary `_addr` allowances created by other contracts.

Resolution

The issue is resolved in commit a34c04d

EGA-02	Revoked Tokens Counted As Surplus Allow Beneficiary To Bypass Revocation		
Asset	`EthgasTokenLock.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

The contract treats revoked tokens as surplus, which allows the beneficiary to withdraw the revoked portion by front-running the admin's `withdrawRevoked()` call, effectively undoing the revocation and giving early access to unvested tokens.

Surplus and outstanding amounts are defined as:

```
function totalOutstandingAmount() public view override returns (uint256) {
    return managedAmount - releasedAmount - revokedAmount;
}
```

```
function surplusAmount() public view override returns (uint256) {
    uint256 balance = currentBalance();
    uint256 outstandingAmount = totalOutstandingAmount();
    if (balance > outstandingAmount) {
        return balance - outstandingAmount;
    }
    return 0;
}
```

The beneficiary can withdraw any positive `surplusAmount`:

```
function withdrawSurplus(uint256 _amount) external override onlyBeneficiary {
    require(_amount > 0, "Amount cannot be zero");
    require(surplusAmount() >= _amount, "Amount requested > surplus available");

    token.safeTransfer(beneficiary, _amount);

    emit TokensWithdrawn(beneficiary, _amount);
}
```

The revoke flow is:

```
function revoke(bytes32[] calldata _ids) external override onlyAdminRole {
    require(revocable, "Contract is non-revocable");
    require(isRevoked == false, "Already revoked");

    uint256 unvestedAmount = managedAmount - vestedAmount();
    require(unvestedAmount > 0, "No available unvested amount");

    revokedAmount = unvestedAmount;
    isRevoked = true;

    emit TokensRevoked(beneficiary, unvestedAmount);
    ...
}
```

```
function withdrawRevoked() external onlyAdminRole {
    token.safeTransfer(msg.sender, revokedAmount);
}
```

A possible scenario:

- Lock is revocable, `managedAmount = 1000` tokens, fully funded at deployment:
  - `currentBalance = 1000`
  - `releasedAmount = 0`
  - `revokedAmount = 0`
  - `totalOutstandingAmount = 1000`
  - `surplusAmount = 0` (no surplus)
- Time passes. Vesting makes `vestedAmount = 300` and `unvestedAmount = 700`. No releases or surplus withdrawals have happened. Balance remains `1000`.
- Admin calls `revoke()`:
  - `unvestedAmount = managedAmount - vestedAmount = 1000 - 300 = 700`
  - `revokedAmount = 700`
  - `totalOutstandingAmount = managedAmount - releasedAmount - revokedAmount = 1000 - 0 - 700 = 300`
  - `currentBalance` is still `1000`
  - `surplusAmount = currentBalance - totalOutstandingAmount = 1000 - 300 = 700`

At this point, `surplusAmount` is exactly equal to `revokedAmount`, even though these tokens are meant to be reserved for the admin. Until the admin successfully calls `withdrawRevoked()`, the beneficiary can:

- Observe the `revoke()` transaction in the mempool.
- Front-run any `withdrawRevoked()` call with:

```
withdrawSurplus(700);
```

Because `surplusAmount() >= 700` holds, the call passes both checks and transfers 700 tokens (the full revoked portion) to the beneficiary. When the admin later calls `withdrawRevoked()`, the contract will still attempt to send `revokedAmount` to the admin, but the underlying balance has been drained by the earlier `withdrawSurplus()` call. In practice the admin will either receive fewer tokens than expected or see the transfer fail if the balance is insufficient.

Revocation does not reliably protect unvested funds. The beneficiary can still obtain the unvested part immediately after a revocation decision, as long as the admin has not yet executed `withdrawRevoked()`. The beneficiary does not need to wait for the full vesting schedule. As soon as at least one vesting period has passed and some amount is considered vested, a revocation followed by `withdrawSurplus()` can give the beneficiary almost the full `managedAmount` early, with only the released or already vested tokens deducted.

Given that revocation is normally the primary control for the admin to reclaim unvested tokens (for example, when an employee leaves early), this behaviour breaks that core guarantee.

Recommendations

Treat revoked tokens as reserved, not surplus, for example by making `surplusAmount()` return zero once `isRevoked == true`, or by ensuring revoked tokens are never included in the surplus calculation.

Resolution

The `revoke()` function was updated to transfer tokens within the same transaction. The issue is resolved in pull/1

EGA-03	Users Can Lose Rewards By Providing Incomplete Token List		
Asset	`EthgasRebate.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Medium	Impact: High	Likelihood: Low

Description

The `claimReward()` function accepts two separate parameters relevant to token distribution:

- `ClaimEntry[] calldata _entries` – contains the full list of user addresses, tokens, and claimable amounts used for Merkle proof verification and internal accounting.
- `address[] calldata _tokens` – a user-supplied list that determines which tokens will actually be processed and transferred/staked during the claim.

There is no validation that `_tokens` contains all distinct token addresses present in `_entries`.

The function proceeds as follows:

1. Marks the entire Merkle root + category as claimed for `msg.sender` immediately (before any transfers).
2. Builds the leaf hash using all `_entries` (correct Merkle verification).
3. When aggregating amounts, it only increments `tokenClaims[j].totalClaimAmount` for tokens that exist in the `_tokens` array.
4. Only tokens present in `_tokens` are later transferred or staked.

As a result, if a user (or frontend) accidentally omits one or more tokens from `_tokens`, the following occurs:

- The Merkle proof is still consumed and the entire reward for that category/root is permanently marked as claimed via: `rewardClaimed[msg.sender][_category][merkleRoot[_category]] = true;`
- Any claim amounts for the missing tokens are effectively lost forever. They are not transferred, and the user can never claim them again.

Recommended Mitigation

Remove the `_tokens` parameter entirely and dynamically compute the unique tokens from `_entries`.

Resolution

The issue was acknowledged by the development team with the following comment.

> *"We don't recommend the user to claim by themselves, we as backend can always update a new merkle root in case any mistake"*

EGA-04	Beneficiary Can Grief Admin By Staking Unvested Tokens		
Asset	`EthgasTokenLock.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

A beneficiary can front-run an admin's revocation attempt by locking all tokens, including unvested amounts, into the `VotingEscrow` contract for up to 4 years, preventing the admin from recovering unvested funds for an extended period.

The `stake()` function allows the beneficiary to stake the full current balance without considering vested/unvested amounts:

```
function stake(uint256 _amount, uint256 _initUnlockTime) external onlyBeneficiary {
    require(_amount <= currentBalance(), "No available balance");
    // @audit No check against vestedAmount() or releasableAmount()
    IVotingEscrow.LockedBalance memory l = veToken.locked(address(this));
    if (l.amount > 0) {
        veToken.increase_amount(_amount);
    } else {
        veToken.create_lock(_amount, _initUnlockTime);
    }
    emit TokensStaked(address(this), _amount);
}
```

This allows the beneficiary to front-run the admin's `revoke()` call by locking all tokens (including unvested amounts) into the `VotingEscrow` contract for up to 4 years.

Attack scenario:

1. Beneficiary vests normally and withdraws vested amounts
2. Admin decides to revoke and recover the remaining unvested tokens
3. Beneficiary front-runs the `revoke()` transaction
4. Beneficiary stakes full balance (including unvested tokens) for 4 years
5. Admin's `revoke()` succeeds and sets `revokedAmount`, but `withdrawRevoked()` fails because tokens are locked in `VotingEscrow`
6. Admin cannot access revoked tokens for up to 4 years

Whilst the tokens are not permanently locked (maximum 4 years), this griefing attack significantly delays the admin's ability to recover unvested funds, potentially impacting business operations or reallocation of tokens to other team members.

Recommendations

Validate staking amount against releasable or vested amounts:

```
function stake(uint256 _amount, uint256 _initUnlockTime) external onlyBeneficiary {
    require(_amount <= releasableAmount(), "Cannot stake more than releasable amount");
    // Or alternatively: require(_amount <= vestedAmount() - releasedAmount, "Cannot stake unvested tokens");

    IVotingEscrow.LockedBalance memory l = veToken.locked(address(this));
    if (l.amount > 0) {
        veToken.increase_amount(_amount);
    } else {
        veToken.create_lock(_amount, _initUnlockTime);
    }
    emit TokensStaked(address(this), _amount);
}
```

Resolution

The staking logic was updated to prevent staking of unvested tokens prior to revocation. The issue is resolved in pull/1

EGA-05	Per-Category Minimum Unlock Duration Not Enforced For Existing Locks		
Asset	`EthgasRebate.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: Low	Likelihood: High

Description

Per-category `minUnlockDuration` is only enforced when creating the first ve lock for a user, so later airdrops with stricter minimum durations can be effectively bypassed by users who already have an older, shorter lock.

When staking rewards, the contract either extends an existing lock or creates a new one based on whether `veToken.locked(msg.sender).amount` is non-zero:

```
if (_isStake && token == address(ethgasToken)) {
    IVotingEscrow.LockedBalance memory l = veToken.locked(msg.sender);
    if (l.amount > 0) {
        veToken.increase_amount_for(msg.sender, totalClaimAmount);
        emit RewardStaked(msg.sender, totalClaimAmount, 0);
    } else {
        if (_initUnlockTime - block.timestamp < uint256(merkleRootInfo[_category].minUnlockDuration)) {
            revert InvalidUnlockTime();
        }
        veToken.create_lock_for(msg.sender, totalClaimAmount, _initUnlockTime);
        emit RewardStaked(msg.sender, totalClaimAmount, _initUnlockTime);
    }
}
```

`minUnlockDuration` is set per category by `updateMerkleRootInfo()`:

```
function updateMerkleRootInfo(bool _isStake, uint248 _minUnlockDuration, bytes32 _category) external onlyBookKeeperRole
    ↪ whenNotPaused {
    // can only update when restricted mode is on
    if (!isRestrictedMode) {
        revert RestrictedModeOff();
    }
    merkleRootInfo[_category] = MerkleRootInfo(_isStake, _minUnlockDuration);
    emit MerkleRootInfoUpdated(_isStake, _minUnlockDuration, _category);
}
```

For the first stake in a category, the contract enforces `_initUnlockTime - block.timestamp >= minUnlockDuration` before calling `create_lock_for()`. For subsequent stakes, if the user already has any lock (`l.amount > 0`), it calls `increase_amount_for()` without checking the current category's `minUnlockDuration`.

This leads to two problematic cases.

First, if a user initially stakes for a short duration under a lenient campaign (category A), and the admin later creates a stricter campaign (category C) with a larger `minUnlockDuration`, the same user can claim category C with stake enabled and have the new tokens added to the old lock, which still unlocks at the original (short) time. The stricter minimum for C is never enforced.

Second, if the admin later reduces `minUnlockDuration` for a new category, new users can lock for a shorter period, while existing stakers remain locked to the old, longer end time. That creates inconsistent and arguably unfair staking terms across users.

Recommendations

Enforce category-specific `minUnlockDuration` even when increasing an existing lock, for example by requiring that the remaining lock time is at least the current category's `minUnlockDuration` before calling `increase_amount_for()`.

Resolution

The staking logic was updated to disallow staking into expired locks and to consistently enforce minimum unlock durations when creating new locks, mitigating the category bypass described. The issue is resolved in pull/1

EGA-06	Front-Running Revoke Allows Beneficiary To Keep Snapshot Voting Power		
Asset	`EthgasTokenLock.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

`revoke()` relies on an `_ids` array supplied by the admin to clear snapshot delegation, which lets a beneficiary front-run the transaction and keep snapshot voting power for the full ve lock duration even after revocation.

```
function revoke(bytes32[] calldata _ids) external override onlyAdminRole {
    require(revocable, "Contract is non-revocable");
    require(isRevoked == false, "Already revoked");

    uint256 unvestedAmount = managedAmount - vestedAmount();
    require(unvestedAmount > 0, "No available unvested amount");

    revokedAmount = unvestedAmount;
    isRevoked = true;

    emit TokensRevoked(beneficiary, unvestedAmount);

    for (uint256 i; i < _ids.length; i++) {
        _clearSnapshotDelegate(_ids[i]);
    }
}
```

Delegation is set and cleared as follows:

```
function setSnapshotDelegate(bytes32 _id, address _delegate) external onlyBeneficiary {
    require(isRevoked == false, "revoked contract cannot perform delegation");
    delegateRegistry.setDelegate(_id, _delegate);
    emit SetDelegate(_id, _delegate);
}
```

```
function _clearSnapshotDelegate(bytes32 _id) internal {
    if (delegateRegistry.delegation(address(this), _id) != address(0)) {
        delegateRegistry.clearDelegate(_id);
        emit ClearDelegate(_id);
    }
}
```

Attack path:

- Beneficiary stakes all tokens for a long period (for example, four years) using `stake()`, does not set any delegate initially and does not call `acceptLock()`.
- Admin observes there is no delegation and calls `revoke()` with an empty `_ids` array.
- The beneficiary front-runs it by calling `setSnapshotDelegate()` with some `_id`, creating a delegation in the registry.
- When `revoke()` executes, it sets `isRevoked = true` but iterates only over the supplied `_ids` (which do not include the new `_id`), so `_clearSnapshotDelegate()` is never called for that id and the existing delegation remains.
- After revocation, `setSnapshotDelegate()` is blocked by `isRevoked`, and only `clearSnapshotDelegate()` (beneficiary-controlled) can remove the delegation. The admin cannot clear it.

In effect, the admin can revoke unvested funds but cannot reliably remove snapshot voting power linked to the long ve lock: a beneficiary that front-runs once can keep voting rights for the full remaining lock, breaking the implied guarantee that a revoked lock stops participating in offchain governance under admin control.

Recommendations

Remove dependence on externally supplied `_ids` and track active delegation ids onchain (for example, store every `_id` used in `setSnapshotDelegate()` in an internal list), then in `revoke()` iterate this internal list and clear all delegations, regardless of when they were created.

Resolution

Staking is restricted to users with a fully vested balance or after revocation, preventing beneficiaries from retaining snapshot voting power tied to unvested tokens after an admin revocation. The issue is resolved in pull/1

EGA-07	Missing Acceptance Check Before Staking Allows Admin To Sweep Returned Funds		
Asset	`EthgasTokenLock.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

The lock can be staked and unstaked before the beneficiary accepts it, allowing an admin to cancel the lock and sweep tokens that have just been returned from ve staking.

`cancelLock()` only checks that `isAccepted` is `false` and then transfers the entire contract balance to the caller:

```
function cancelLock() external onlyAdminRole {
    require(isAccepted == false, "Cannot cancel accepted contract");

    token.safeTransfer(msg.sender, currentBalance());

    emit LockCanceled();
}
```

However, `stake()` and `unstake()` are not gated by `isAccepted`:

```
function stake(uint256 _amount, uint256 _initUnlockTime) external onlyBeneficiary {
    require(_amount <= currentBalance(), "No available balance");
    IVotingEscrow.LockedBalance memory l = veToken.locked(address(this));
    if (l.amount > 0) {
        veToken.increase_amount(_amount);
    } else {
        veToken.create_lock(_amount, _initUnlockTime);
    }
    emit TokensStaked(address(this), _amount);
}
```

```
function unstake() external {
    veToken.withdraw();
    emit TokensUnstaked();
}
```

A realistic sequence is:

- Beneficiary stakes funds via `stake()` for a period, but does not call `acceptLock()`.
- Once the ve lock expires, any account calls `unstake()`, sending the tokens back to the lock contract.
- Because `isAccepted` is still `false`, an admin can call `cancelLock()` and sweep the full `currentBalance`, including the tokens that have just been unstaked.

Recommendations

- Require `isAccepted == true` in `stake()` and `unstake()`, so staking activity is only possible after the beneficiary has committed to the lock; or
- Modify `cancelLock()` to revert if there has been staking activity (for example if `veToken.locked(address(this)).amount > 0` or a dedicated `hasStaked` flag is set).

Resolution

The issue is resolved in commit 4cf0bd7

EGA-08	Inconsistent Managed Amount And Period Configuration Causes Dust Amounts		
Asset	`EthgasTokenLock.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Medium	Impact: Low	Likelihood: High

Description

Lack of consistency checks between `managedAmount` and the configured period counts can lead to "dust" that is never included in the per-period vesting and unlock schedule, causing slightly inconsistent payouts and residual surplus.

`managedAmount` is set directly from `_managedAmount` with no relation enforced to `vestingPeriods` or `unlockPeriods`:

```
aclManager = _aclManager;
beneficiary = _beneficiary;
token = _token;
veToken = _veToken;
feeDistributor = _feeDistributor;
delegateRegistry = _delegateRegistry;

managedAmount = _managedAmount;
```

Per-period amounts are computed using integer division:

```
function vestingAmountPerPeriod() public view override returns (uint256) {
    return (managedAmount - vestingCliffAmount) / vestingPeriods;
```

```
function unlockAmountPerPeriod() public view override returns (uint256) {
    return (managedAmount - initialUnlockAmount) / unlockPeriods;
```

If `(managedAmount - vestingCliffAmount)` or `(managedAmount - initialUnlockAmount)` are not exactly divisible by the respective period counts, the remainder is silently dropped. This leaves a small residual amount that is never part of the scheduled vesting/unlock stream and only appears as surplus via `surplusAmount`, which may be surprising for users expecting all of `managedAmount` to be distributed evenly across periods.

Recommendations

Validate that the configured `managedAmount` is consistent with the vesting and unlock period configuration, or otherwise account for the remainder explicitly.

Resolution

Any rounding remainder is caught in the final release (or on revoke), so the full `managedAmount` is always delivered rather than left as dust.

EGA-09	Staking Rewards For Vested Tokens Can Be Claimed By Admin After Revocation		
Asset	`EthgasTokenLock.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Medium

Description

`claimStakingReward()` does not distinguish between rewards earned on vested tokens and rewards earned on unvested tokens, which means that after revocation the admin can receive rewards that are economically tied to the beneficiary's vested stake.

```
function claimStakingReward() external {
    require(currentTime() >= unlockStartTime, "cannot claim before unlockStartTime");
    uint256 initBalance = currentBalance();
    feeDistributor.claim(address(this));
    uint256 endBalance = currentBalance();
    uint256 rewardToRelease = endBalance - initBalance;
    if (rewardToRelease == 0) {
        return;
    }
    if (!isRevoked) {
        token.safeTransfer(beneficiary, rewardToRelease);
        emit RewardClaimed(beneficiary, rewardToRelease);
    } else {
        // only admin can claim reward for revoked contract, even revoked amount is less than staked amount
        aclManager.checkAdminRole(msg.sender);
        token.safeTransfer(msg.sender, rewardToRelease);
        emit RewardClaimed(msg.sender, rewardToRelease);
    }
}
```

There are two staking routes:

- via `release(_isStake = true)`, where already vested tokens are staked on behalf of the beneficiary, and
- via `stake()`, where tokens in the lock (which may include unvested amounts) are staked under the contract itself.

When `isRevoked` becomes `true`, all subsequent rewards claimed through `claimStakingReward()` are sent to the admin, even if part of the underlying stake corresponds to previously vested tokens that should be treated as the beneficiary's property. This creates an unfair allocation where revocation can redirect some vested rewards to the admin.

Recommendations

- Track rewards or staked balances attributable to vested and unvested tokens separately, and after revocation only redirect to the admin the portion linked to unvested (revoked) tokens; or
- Alternatively, always route rewards from any stake that used vested tokens to the beneficiary, regardless of revocation state.

Resolution

Staking rewards are now always transferred to the beneficiary, preventing misallocation of rewards derived from vested tokens after revocation. The issue is resolved in pull/1

EGA-10	Division By Zero In Vesting View Functions When Revocable Is False		
Asset	`EthgasTokenLock.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

When a contract is deployed with `revocable = false`, multiple view functions revert with division by zero errors, breaking external integrations, UIs, and monitoring tools that query contract state.

When a contract is deployed with `revocable = false`, the vesting-related variables are never initialised in the constructor. They remain at their default values of 0:

```
if (_revocable) {
    // Only set these if revocable = true
    vestingPeriods = _vestingInfo.vestingPeriods;
    vestingCliffTime = _vestingInfo.vestingCliffTime;
    vestingEndTime = _vestingInfo.vestingEndTime;
    vestingCliffAmount = _vestingInfo.vestingCliffAmount;
    revocable = _revocable;
}
// If revocable = false, all vesting variables remain 0
```

This causes multiple view functions to revert with division by zero errors:

- `vestingAmountPerPeriod()` (line [**249**]): Divides by `vestingPeriods` (0)
- `vestingPeriodDuration()` (line [**257**]): Divides by `vestingPeriods` (0)
- `currentVestingPeriod()` (line [**265**]): Calls `vestingPeriodDuration()` which reverts
- `passedVestingPeriods()` (line [**273**]): Calls `currentVestingPeriod()` which reverts

Additionally, `passedVestingPeriods()` (line [**273**]) would underflow (`currentVestingPeriod()` returns 1 from `MIN_PERIOD`, minus `MIN_PERIOD` tries to go negative).

For non-revocable contracts, these functions should return sensible defaults (0 or managed amount) rather than reverting, as they may be called by external integrations, UIs, or monitoring tools.

Recommendations

Add checks for non-revocable contracts in vesting functions to return appropriate default values rather than attempting division by zero or calling functions that will revert.

Resolution

The issues have been resolved in commit 0718b21

EGA-11	Integer Overflow In Unlock Time Validation		
Asset	`EthgasRebate.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The unlock time validation performs a subtraction that can negatively overflow in Solidity 0.8.x, causing confusing panic errors instead of the intended custom error when users provide past or present timestamps.

At line [223], the validation check performs a subtraction that can overflow if `_initUnlockTime` is less than or equal to `block.timestamp`:

```
if (_initUnlockTime - block.timestamp < uint256(merkleRootInfo[_category].minUnlockDuration)) {
    revert InvalidUnlockTime();
}
```

In Solidity `0.8.x`, this will revert with a panic error instead of the intended `InvalidUnlockTime` error, making it harder to diagnose the issue. More importantly, if the user provides an unlock time in the past or present, the transaction fails.

Recommendations

Add explicit validation before the subtraction:

```
if (_initUnlockTime <= block.timestamp) {
    revert InvalidUnlockTime();
}
if (_initUnlockTime - block.timestamp < uint256(merkleRootInfo[_category].minUnlockDuration)) {
    revert InvalidUnlockTime();
}
```

Resolution

The issues is resolved in commit 0718b21

EGA-12	Missing Validation For Constructor Amounts		
Asset	`EthgasTokenLock.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The constructor fails to validate that `vestingCliffAmount` and `initialUnlockAmount` do not exceed `managedAmount`, which could cause underflow reverts in vesting and unlocking calculations.

The constructor at does not validate that `vestingCliffAmount` and `initialUnlockAmount` are less than or equal to `managedAmount`:

```
constructor(
    // ... parameters
) {
    // ... other validations

    managedAmount = _managedAmount;

    unlockPeriods = _unlockInfo.unlockPeriods;
    unlockStartTime = _unlockInfo.unlockStartTime;
    unlockEndTime = _unlockInfo.unlockEndTime;
    initialUnlockAmount = _unlockInfo.initialUnlockAmount; // @audit No validation

    if (_revocable) {
        // ... other validations
        vestingCliffAmount = _vestingInfo.vestingCliffAmount; // @audit No validation
        revocable = _revocable;
    }
}
```

If `vestingCliffAmount > managedAmount` or `initialUnlockAmount > managedAmount`, the calculations in vesting/unlocking functions could underflow or produce incorrect results:

- `(managedAmount - vestingCliffAmount)` would underflow if `vestingCliffAmount > managedAmount`
- `(managedAmount - initialUnlockAmount)` would underflow if `initialUnlockAmount > managedAmount`

Recommendations

Add validation in constructor to ensure amounts do not exceed managed amount.

Resolution

The issue is resolved in commit 0718b21

EGA-13	Beneficiary-Controlled Rotation Allows Pre-Change Draining Of Lock		
Asset	`EthgasTokenLock.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

Allowing the current beneficiary to unilaterally update the `beneficiary` address introduces a safety concern where a beneficiary can drain funds before handing over the lock, undermining the apparent intent of changing ownership.

The `changeBeneficiary()` function is restricted to the current `beneficiary`:

```
function changeBeneficiary(address _newBeneficiary) external onlyBeneficiary {
    require(_newBeneficiary != address(0), "Empty beneficiary");
    beneficiary = _newBeneficiary;
    emit BeneficiaryChanged(_newBeneficiary);
}
```

In practice, if the lock represents compensation or a long-term allocation for an employee, the employee (as current beneficiary) can first fully exercise their rights (e.g. release any available tokens, stake and claim rewards where possible) and then call `changeBeneficiary()` to set a successor. From the new beneficiary's perspective, this may appear as a transfer of a lock while in reality the previous beneficiary has already extracted all value allowed by the schedule.

Recommendations

Restrict or remove beneficiary self-rotation to avoid misleading handovers and rely on explicit administrator-controlled flows instead.

Gate `changeBeneficiary()` by `onlyAdminRole` (optionally with a timelock or dual consent pattern) so that beneficiary changes are an explicit, governed action rather than controlled solely by the outgoing beneficiary.

Resolution

The issue was acknowledged by the development team with the following comment.

> *"new beneficiary will be a new address for the old beneficiary"*

EGA-14	Missing View Functions For Contract State
Asset	`EthgasTokenLock.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The contract lacks convenient view functions to query important state information about staked tokens, making it difficult for users, UIs, and monitoring tools to understand the full contract state without manual calculations.

The contract lacks convenient view functions to query important state information:

1. **Staked amount**: No function to check how many tokens from `managedAmount` are currently staked in VotingEscrow
2. **Available balance excluding staked**: No function to see liquid balance in contract (current balance minus staked amount)
3. **Staked balance details**: No way to query the lock details (amount and end time) for the contract's stake

Users, UIs, and monitoring tools would benefit from these convenience functions:

```
// Current implementation only has:
function currentBalance() public view returns (uint256) {
    return token.balanceOf(address(this));
}

// Missing functions like:
function stakedBalance() public view returns (uint256);
function availableLiquidBalance() public view returns (uint256);
function getStakeLockInfo() public view returns (uint256 amount, uint256 endTime);
```

Recommendations

Consider adding convenience view functions that provide information about staked balances, available liquid balances, and stake lock details.

Resolution

The issue is resolved in commit 0718b21

EGA-15	Unused SmartWalletChecker Interface
Asset	`VotingEscrow.vy`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The `VotingEscrow` contract contains an unused interface definition for `SmartWalletChecker` that is never referenced, adding unnecessary code clutter and potential confusion about the whitelist implementation.

At line [**47-53**], there is an interface definition for `SmartWalletChecker`:

```
# Interface for checking whether address belongs to a whitelisted
# type of a smart wallet.
# When new types are added - the whole contract is changed
# The check() method is modifying to be able to use caching
# for individual wallet addresses
interface SmartWalletChecker:
    def check(addr: address) -> bool: nonpayable
```

This interface is never used anywhere in the contract. The contract uses a simpler `whitelisted_contracts` mapping instead (line [**113**]) with direct boolean checks.

This is leftover code from the original Curve Finance implementation that has not been cleaned up in this fork.

Recommendations

Consider removing the unused interface definition to reduce code clutter and prevent confusion about the whitelist implementation.

Resolution

The issue is resolved in a34c04d

EGA-16	Missing Reentrancy Guard On Deposit Functions
Asset	`EthgasRebate.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The `deposit()` function lacks reentrancy protection, whereas other state-changing functions in the contract utilise the `nonReentrant` modifier, creating an inconsistency in the security model.

The `deposit()` functions lack reentrancy protection, whilst the `claimReward()` function has a `nonReentrant` modifier:

```
function deposit() payable external onlyDepositor whenNotPaused {
    if(msg.value > 0) {
        IWETH(weth).deposit{value: msg.value}();
        emit Deposit(address(weth), msg.sender, msg.value);
    } else {
        revert InvalidDepositAmount();
    }
}

function deposit(address[] calldata _tokenAddr, uint256[] calldata _amount) external onlyDepositor whenNotPaused {
    if (_tokenAddr.length != _amount.length) {
        revert InvalidArrayLength();
    }
    for (uint256 i; i < _tokenAddr.length; i++) {
        IERC20(_tokenAddr[i]).safeTransferFrom(msg.sender, address(this), _amount[i]);
        emit Deposit(_tokenAddr[i], msg.sender, _amount[i]);
    }
}
```

Whilst the `onlyDepositor` modifier restricts access to trusted parties, and standard WETH implementations are safe, the lack of reentrancy protection represents an inconsistency with the rest of the contract's security model.

Recommendations

Add `nonReentrant` modifier to both deposit functions for consistency with the contract's security model.

Resolution

The issue is resolved in commit 0718b21

EGA-17	Miscellaneous General Comments
Asset	All contracts
Status	**Resolved:** See Resolution
Rating	Informational

Description

This section details miscellaneous findings discovered by the testing team that do not have direct security implications:

1. **Missing Event Emission In `withdrawRevoked()`**

   Related Asset(s): EthgasTokenLock.sol

   The `withdrawRevoked()` function transfers tokens without emitting any event, making it difficult to track administrative withdrawals through onchain monitoring, indexing services, or frontend applications.

   The `withdrawRevoked()` function at 527-531 transfers tokens but does not emit any event:

   ```
   function withdrawRevoked() external onlyAdminRole {
       token.safeTransfer(msg.sender, revokedAmount);
   }
   ```

   All other token transfer functions in the contract emit events:

   - `release()` emits `TokensReleased` (line [426])
   - `withdrawSurplus()` emits `TokensWithdrawn` (line [502])
   - `cancelLock()` emits `LockCanceled` (line [200])

   Events are crucial for:

   - Offchain monitoring and indexing
   - Audit trails
   - Frontend applications tracking contract state
   - Analytics and reporting

   The lack of an event makes it difficult to track when and how much revoked tokens were withdrawn by admins.

   Consider emitting an event when withdrawing revoked tokens, either by reusing the existing `TokensWithdrawn` event or defining a more specific `RevokedTokensWithdrawn` event.

2. **Typos In Event Name And Comments**

   Related Asset(s): EthgasRebate.sol

   The codebase contains the following typos:

   (a) At `EthgasRebate.sol:125`, the event name has a typo:

   ```
   - emit DepositWhitelistStatusChagned(_depositorAddr[i], _status[i]);
   + emit DepositWhitelistStatusChanged(_depositorAddr[i], _status[i]);
   ```

   (a) At `VotingEscrow.vy:231`, there is a typo in the comment:

```
- @param old_locked Pevious locked amount / end lock time for the user
+ @param old_locked Previous locked amount / end lock time for the user
```

(a) At `EthgasTokenLock.sol:233`, there is a typo in the NatSpec comment:

```
- @dev Returns zero if called before conctract starTime
+ @dev Returns zero if called before contract startTime
```

(a) At `EthgasTokenLock.sol:508`, there is a typo in the NatSpec comment:

```
- @param _ids has no effect if there hasn't benn any snapshot delegation
+ @param _ids has no effect if there hasn't been any snapshot delegation
```

Consider fixing the typos mentioned above.

3. **Misleading Comment**
   Related Asset(s): FeeDistributor.vy
   The natspec for `whitelist_claimer()` is misleading as it says it whitelisting for `claim_and_stake()`, however it is actually a whitelisting for `claim_many_and_stake()`.

Recommendations

Ensure that the comments are understood and acknowledged, and consider implementing the suggestions above.

Resolution

The issues are resolved in commit 0718b21 and commit 235cabd

Appendix A Vulnerability Severity Classification

This security review classifies vulnerabilities based on their potential impact and likelihood of occurance. The total severity of a vulnerability is derived from these two metrics based on the following matrix.

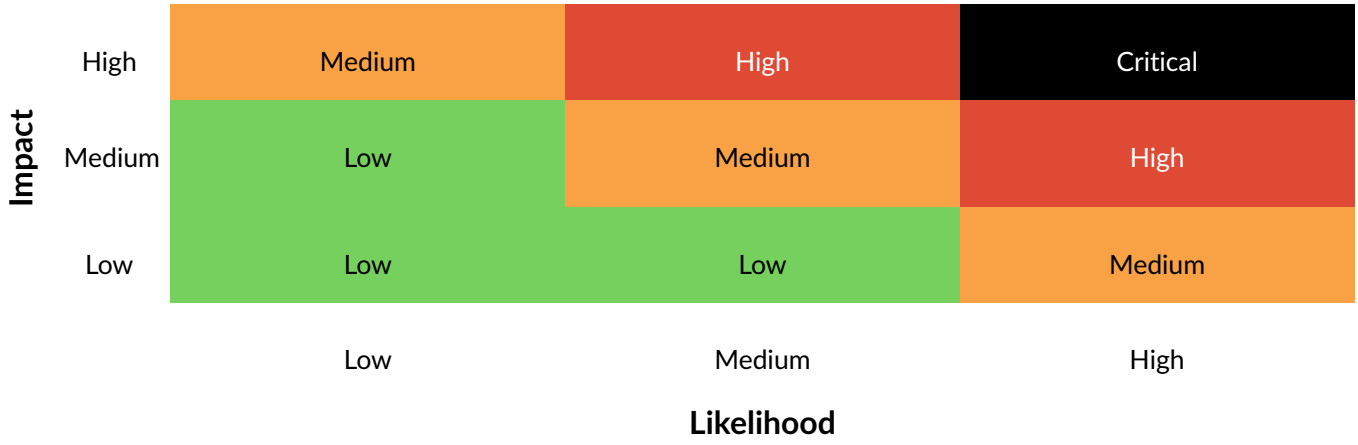

Table 1: Severity Matrix - How the severity of a vulnerability is given based on the *impact* and the *likelihood* of a vulnerability.