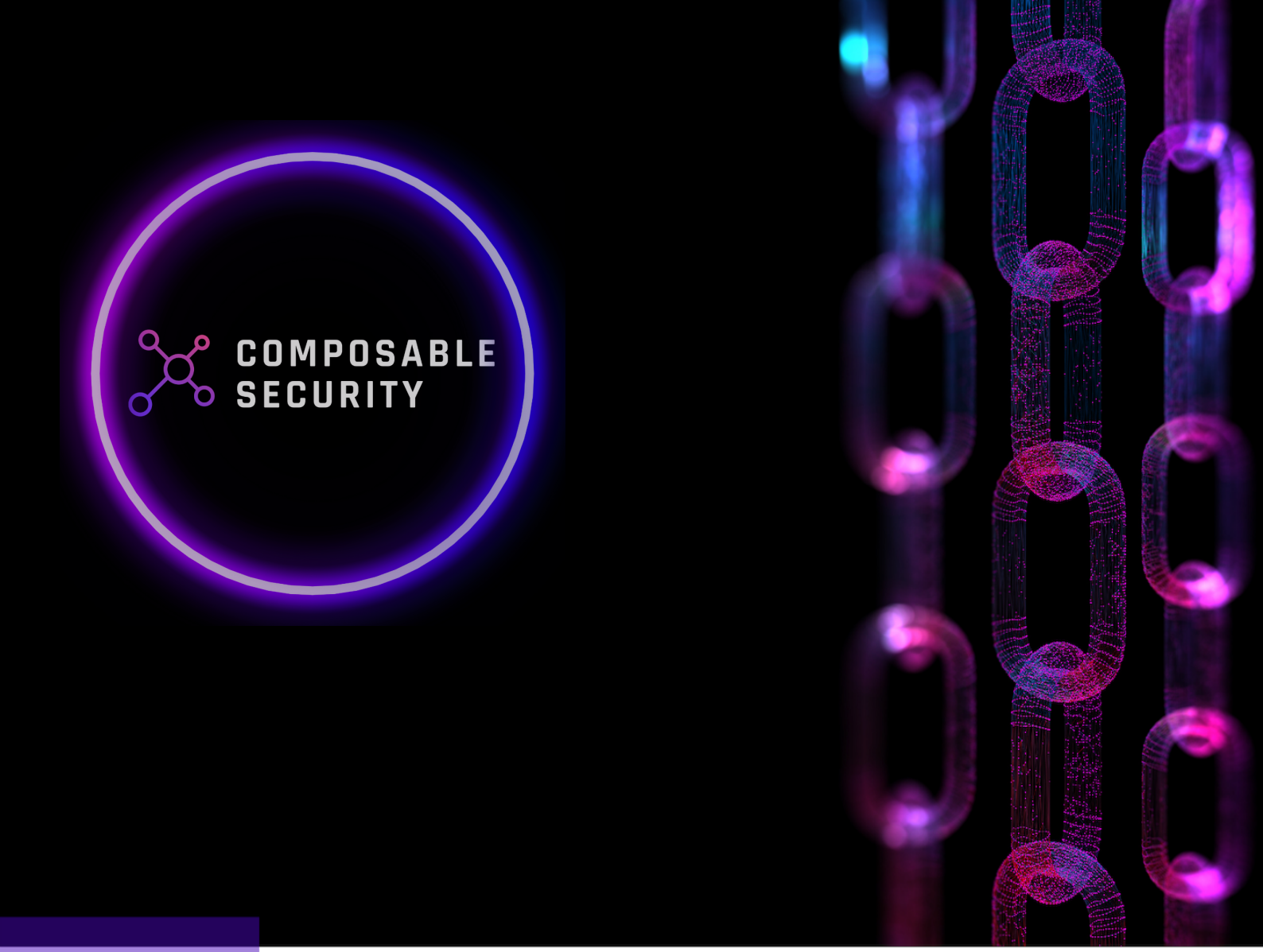


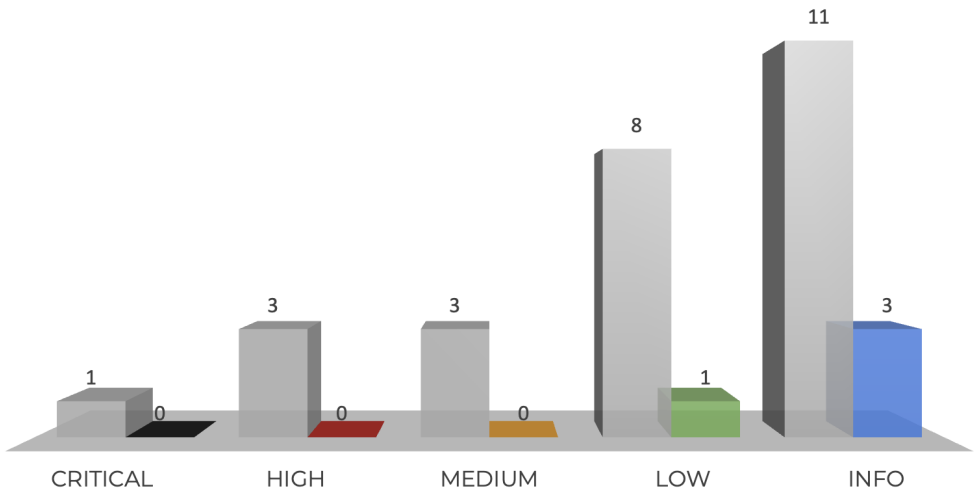
REPORT

Smart contract security review for Neverland Money

Prepared by: Composable Security
Report ID: NRL-6c19a5e
Test time period: 2025-07-14 - 2025-07-25
Retest time period: 2025-08-14 - 2025-08-22
Report date: 2025-08-22
Version: 1.1
Visit: composable-security.com

Contents

1. Retest summary (2025-08-22)



The description of the current status for each retested vulnerability and recommendation has been added in its section.

1.1. Results

The **Composable Security** team participated in a one-time iteration to verify if the vulnerabilities detected during the tests (between 2025-07-14 and 2025-07-25) were correctly removed and no longer appear in the code.

The current status of the detected issues is as follows:

- 1 **critical** vulnerability has been removed from the code.
- 3 **high** vulnerabilities have been removed from the code.
- 3 vulnerabilities with a **medium** impact on risk have been removed from the code.
- 8 vulnerabilities with a **low** impact on risk were handled as follows:
  - 1 have been acknowledged,
  - 7 have been fixed.
- 11 security **recommendations** were handled as follows:
  - 8 have been implemented,
  - 3 have been acknowledged.

The team addressed all detected high-risk vulnerabilities and implemented most of the recommendations. They were very involved in the vulnerability removal process.

The team addressed the vulnerabilities over several iterations. Since early attempts did not completely resolve the issues, the final, comprehensive fixes were verified in later commits.

One of the vulnerabilities was downgraded to low after further discussion with the team.

Due to the number of findings in a given time and the numerous changes made to remove detected vulnerabilities, there is a greater probability that issues are still present in the code, therefore another iteration of the audit should be considered.

1.2. Scope

The retest scope included the same contracts, on a different commit in the same repository.

GitHub repository: https://github.com/Neverland-Money/neverland-contracts
CommitID: 816c394500f185e0d403ef2d8bc94f38ea1dc22d

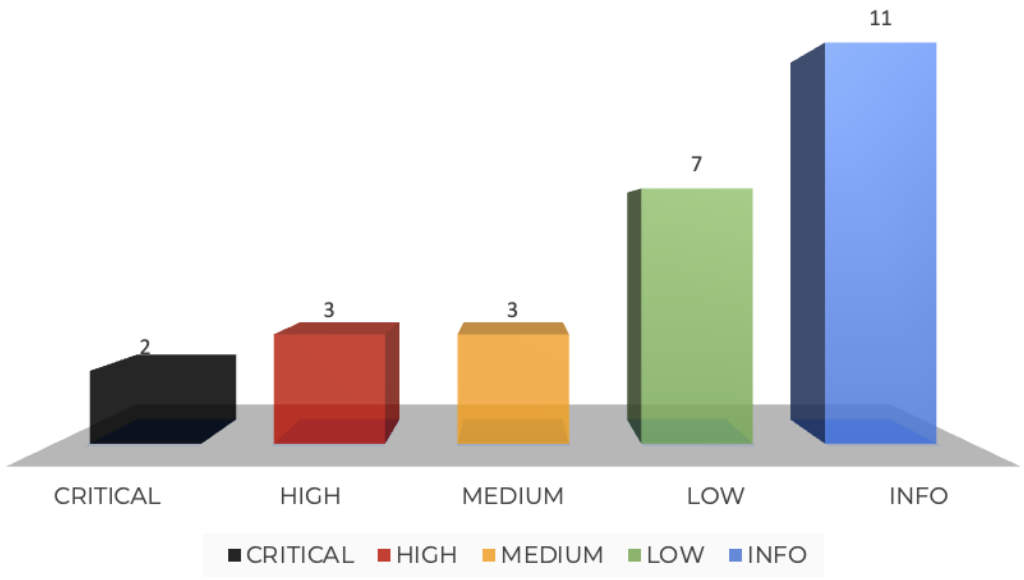
2. Current findings status

ID	Severity	Vulnerability	Status
NRL-6c19a5e-C01	CRITICAL	Early withdrawal penalty fee mechanism bypass	FIXED
NRL-6c19a5e-H01	HIGH	Invalid update of voting power	FIXED
NRL-6c19a5e-H02	HIGH	Integer division truncation in reward calculation	FIXED
NRL-6c19a5e-H03	HIGH	Permanent reward loss for burnt veNFTs	FIXED
NRL-6c19a5e-M01	MEDIUM	Self-repaying loan reward receiver persists across veNFT transfers	FIXED
NRL-6c19a5e-M02	MEDIUM	Missing permanent lock validation in split function	FIXED
NRL-6c19a5e-M03	MEDIUM	Inability to delegate claim permissions by users	FIXED
NRL-6c19a5e-L01	LOW	One-step ownership transfer in setTeam	FIXED
NRL-6c19a5e-L02	LOW	DustLock NFT can be transferred to address 0	FIXED
NRL-6c19a5e-L03	LOW	Non-compliant ERC721 totalSupply implementation	ACKNOWLEDGED
NRL-6c19a5e-L04	LOW	Single-step ownership transfer in upgradeable contract	FIXED
NRL-6c19a5e-L05	LOW	Missing minimum transaction amount validation enables network congestion through dust attacks	FIXED
NRL-6c19a5e-L06	LOW	Missing ownership validation in reward claiming leads to accidental token loss	FIXED
NRL-6c19a5e-L07	LOW	Precision loss in voting power calculation due to division before multiplication	FIXED
NRL-6c19a5e-L08	LOW	Inability to collect any rewards	FIXED

ID	Severity	Recommendation	Status
NRL-6c19a5e-R01	INFO	Do not use assert	IMPLEMENTED

NRL-6c19a5e-R02	INFO	Consider using newest Solidity version	ACKNOWLEDGED
NRL-6c19a5e-R03	INFO	Consider validating the voting state for locks before splitting and merging	ACKNOWLEDGED
NRL-6c19a5e-R04	INFO	Use named constants instead of magic numbers	IMPLEMENTED
NRL-6c19a5e-R05	INFO	Make error message formatting consistent	IMPLEMENTED
NRL-6c19a5e-R06	INFO	Add zero-address validation in critical functions	IMPLEMENTED
NRL-6c19a5e-R07	INFO	Remove redundant decimals	IMPLEMENTED
NRL-6c19a5e-R08	INFO	Verify zero-address rewards receiver in RevenueReward	IMPLEMENTED
NRL-6c19a5e-R09	INFO	Fix value emitted in event	IMPLEMENTED
NRL-6c19a5e-R10	INFO	Require minimum lock amount	ACKNOWLEDGED
NRL-6c19a5e-R11	INFO	Fix naming convention	IMPLEMENTED

3. Security review summary (2025-07-25)



3.1. Client project

The **Neverland Money** project is a lending protocol built on Monad. It focuses on combining AAVE V3's lending functionalities with with proprietary vote-escrowed tokenomics, introducing features such as governance via veDUST, along with unique rewards distribution, emission mechanisms, advanced automated yield strategies, and self-repaying loans.

The audit's primary focus is on custom-developed features, excluding the forked code from AAVE and Velodrome.

3.2. Results

The **Neverland Money** engaged Composable Security to review the security of **Neverland Money**. Composable Security team conducted this assessment over 2 weeks with contractors (Bloqarl, 0xluk3).

The summary of findings is as follows:

- 2 vulnerabilities with a **critical** impact on risk were identified. Their potential consequences are:
  - Permanent lock of all revenue rewards (NRL-6c19a5e-C01).
  - Revenue loss to the protocol treasury (NRL-6c19a5e-C02).
- 3 vulnerabilities with a **high** impact on risk were identified. Their potential consequences are:
  - Inaccurate update of the current total voting power (`bias`) and the adjustments to slopeChanges (NRL-6c19a5e-H01).

    - Unfair distribution favoring large holders (NRL-6c19a5e-H02).
    - Fund leakage and user financial loss (NRL-6c19a5e-H03).
- 3 vulnerabilities with a **medium** impact on risk were identified.
- 7 vulnerabilities with a **low** impact on risk were identified.
- 11 **recommendations** have been proposed that can improve overall security and help implement best practice.
- The team was engaged and the communication was very good.
- The audit has highlighted some serious vulnerabilities. This should be treated as an opportunity to harden the system's security, which is a typical outcome for an initial review. To be thorough, we advise a second audit after the retest. In addition, incorporating security checklists into your peer review process is a best practice that will help maintain a strong security posture over the long term.

Composable Security recommends that **Neverland Money** complete the following:

- Address all reported issues.
- Extend unit tests with scenarios that cover detected vulnerabilities where possible.
- Consider whether the detected vulnerabilities may exist in other places (or ongoing projects) that have not been detected during engagement.
- Review dependencies and upgrade to the latest versions.
- Incorporate security checklists into your peer review process.
- Consider a second audit after the retest.

3.3. Scope

The scope of the tests included selected contracts from the following repository.

GitHub repository: https://github.com/Neverland-Money/neverland-contracts
CommitID: 6c19a5e023c68fb5bdb95389a316e2a6f943a0c6

The detailed scope of tests can be found in Agreed scope of tests.

4. Project details

4.1. Projects goal

The Composable Security team focused during this audit on the following:

- Perform a tailored threat analysis.
- Ensure that smart contract code is written according to security best practices.
- Identify security issues and potential threats both for **Neverland Money** and their users.
- The secondary goal is to improve code clarity and optimize code where possible.

4.2. Agreed scope of tests

The subjects of the test were selected contracts from the **Neverland Money** repository.

GitHub repository:
https://github.com/Neverland-Money/neverland-contracts
CommitID: 6c19a5e023c68fb5bdb95389a316e2a6f943a0c6

Files in scope:

```
.
└── src
    ├── emissions
    │   ├── DustLockTransferStrategy.sol
    │   ├── DustRewardsController.sol
    │   └── DustTransferStrategyBase.sol
    ├── libraries
    │   ├── BalanceLogicLibrary.sol
    │   ├── EpochTimeLibrary.sol
    │   └── SafeCastLibrary.sol
    ├── rewards
    │   └── RevenueReward.sol
    └── tokens
        ├── Dust.sol
        └── DustLock.sol
```

Documentation:

- Scope and brief description
- I got no time for in-depth documentation version
- In-depth documentation

4.3. Threat analysis

This section summarizes the potential threats that were identified during initial threat modeling performed before the audit. The tests were focused, but not limited to, finding security issues that could be exploited to achieve these threats.

Key assets that require protection:

- Users' funds.
- Rewards.
- Protocol revenue.
- Voting power.

Potential attackers goals:

- Theft of user's funds.
- Influencing/increasing voting power.
- Unfair reward distribution.
- Permanent lock of all revenue rewards.
- Lock users' funds in the contract.
- Block the contract, so that others cannot use it.

Potential scenarios to achieve the indicated attacker's goals:

- Influence or bypass the business logic of the system.
- Take advantage of arithmetic errors.
- Privilege escalation through incorrect access control to functions or poorly written modifiers.
- Existence of known vulnerabilities (e.g., front-running, re-entrancy).
- Design issues.
- Excessive power, too much in relation to the declared one.
- Poor security against taking over the managing account.
- Private key compromise, rug-pull.
- Withdrawal of more funds than expected.

4.4. Testing methodology

Smart contract security review was performed using the following methods:

- Q&A sessions with the **Neverland Money** development team to thoroughly understand intentions and assumptions of the project.
- Initial threat modeling to identify key areas and focus on covering the most relevant scenarios based on real threats.
- Automatic tests using slither.
- Custom scripts (e.g. unit tests) to verify scenarios from initial threat modeling.

- **Manual review of the code.**

4.5. Disclaimer

Smart contract security review **IS NOT A SECURITY WARRANTY**.

During the tests, the Composable Security team makes every effort to detect any occurring problems and help to address them. However, it is not allowed to treat the report as a security certificate and assume that the project does not contain any vulnerabilities. Securing smart contract platforms is a multi-stage process, starting from threat modeling, through development based on best practices, security reviews and formal verification, ending with constant monitoring and incident response.

Therefore, we encourage the implementation of security mechanisms at all stages of development and maintenance.

5. Vulnerabilities

[NRL-6c19a5e-C01] Early withdrawal penalty fee mechanism bypass

CRITICAL FIXED

Retest (2025-08-14)

The vulnerability has been completely resolved. The team updated the early withdrawal penalty calculation to use `_balanceOfNFTAt(tokenId, block.timestamp)` instead of the flash-protected `balanceOfNFT()` function, removing the bypass vectors that allowed users to avoid penalties through same-block NFT transfers.

Affected files

- DustLock.sol#L788
- DustLock.sol#L978
- DustLock.sol#L204-L223

Description

The DustLock contract's early withdrawal penalty calculation uses `balanceOfNFT()` which implements flash protection by returning 0 when a token is transferred in the same block. This creates a critical vulnerability where users can completely bypass the 50% early withdrawal penalty by transferring their veNFT and immediately calling `earlyWithdraw()` within the same block.

The vulnerability occurs because the penalty calculation `earlyWithdrawPenalty * balanceOfNFT(_tokenId)/ 10_000` relies on voting power rather than the actual locked token amount. When `balanceOfNFT()` returns 0 due to flash protection, the penalty becomes zero regardless of the actual locked value.

Vulnerable scenario

The following steps lead to the described result:

① User has a veNFT with substantial locked tokens (e.g., 10,000 DUST tokens worth $5,000)
② User wants to exit early to avoid market conditions (normally requiring 50% penalty = $2,500)

③ User transfers the NFT to another address (either self-transfer or to an accomplice) in block N
④ This sets `ownershipChange[tokenId] = block.number` triggering flash protection
⑤ In the same block N, the new owner calls `earlyWithdraw(tokenId)`
⑥ During penalty calculation: `userPenaltyAmount = earlyWithdrawPenalty * balanceOfNFT(tokenId)/ 10_000`
⑦ `balanceOfNFT(tokenId)` returns 0 due to flash protection, making penalty = 0
⑧ The user receives the full $5,000 instead of $2,500, treasury loses $2,500 in penalty fees
⑨ This works with any transfer method: self-transfer, transfer to accomplice, or transfer with subsequent approval

Result: Users can withdraw their entire locked token balance without paying any penalty fees, causing substantial revenue loss to the protocol treasury and completely undermining the penalty mechanism.

Recommendation

Use actual locked amount for penalty calculation instead of voting power:

```
// Replace line 788:
// uint256 userPenaltyAmount = earlyWithdrawPenalty * balanceOfNFT(_tokenId) /
    10_000;

// With:
uint256 userPenaltyAmount = earlyWithdrawPenalty * userLockedAmount / 10_000;
```

This fix eliminates the flash protection interference entirely by basing penalties on the actual stored locked amount rather than calculated voting power.

References

1. SCSVS G4: Business Logic

[NRL-6c19a5e-H01] Invalid update of voting power

HIGH FIXED

Retest (2025-08-14)

The vulnerability has been completely resolved. The team fixed the voting power re-calculation issue by re-evaluating `oldLocked` after calling `unlockPermanent()` in the `earlyWithdraw` function (line 868).

Affected files

- DustLock.sol#L772-L802

Description

The `earlyWithdraw` function invokes the `unlockPermanent` function when the token being withdrawn is classified as a permanent lock. However, it fails to update the `oldLocked` variable after this operation. As a result, the subsequent call to the `_checkpoint` function at line 795 references an outdated version of the token's `locked` state.

Result: Inaccurate update of the current total voting power (`bias`) and the adjustments to `slopeChanges`.

Recommendation

When a permanent lock is processed and the `unlockPermanent` function is executed, the previous locked state should be re-evaluated. The following code adjustment is suggested:

```
if (oldLocked.isPermanent) {
  unlockPermanent(_tokenId);
  oldLocked = _locked[_tokenId];
}
```

References

1. SCSVS G4: Business logic

[NRL-6c19a5e-H02] Integer division truncation in reward calculation

HIGH FIXED

Retest (2025-08-22)

The vulnerability has been completely resolved. The team scale the remainder relative to the supply of the epoch in which it was generated, and accumulate it in scaled form. Once enough is collected, it converts into a full unit. This way, reward distribution remains accurate even as supply fluctuates. They also ensured `lastEarnTime` always updates on claims so that old epochs cannot be counted twice. All relevant functions (preview, claim, split, merge, transfer, burn) correctly propagate or settle frac-

tional rewards. No missed update found. The fix was verified on a separate commits *052c3bebd083b4d47111f1cb7c01fe8d5e1e3864*.

Affected files

- RevenueReward.sol#L135-L138

Description

The `earned()` function uses integer division to calculate proportional rewards for veNFT holders, which systematically truncates fractional amounts and causes permanent reward loss for users with smaller stakes. The calculation `(tokenRewardsPerEpoch[token][_currTs] * dustLock.balanceOfNFTAt(tokenId, _currTs))/ tokenSupplyBalanceCurrTs` rounds down any fractional results to zero due to Solidity's integer arithmetic behavior.

This creates an unfair distribution system where users with small veNFT balances consistently receive zero rewards even when they have legitimately earned fractional amounts, while larger holders capture disproportionately more rewards. The truncated amounts are permanently lost rather than accumulated or redistributed, leading to protocol fund leakage and systematic disadvantage for smaller participants.

Vulnerable scenario

The following steps lead to the described result:

Accidental Loss Scenario:

① User owns veNFT #123 and accumulates small rewards over Weeks 1-3 that are below precision thresholds
② User calls `getReward()` in Week 4
③ `earned()` calculates rewards but returns 0 due to integer division rounding down small amounts
④ `lastEarnTime` is updated to Week 4 timestamp despite no transfer occurring
⑤ User's Week 1-3 rewards are permanently lost as future `earned()` calls start from Week 4

Griefing Attack Scenario:

① Attacker identifies active veNFTs belonging to other users with small stakes
② Attacker repeatedly calls `getReward(victimTokenId, [rewardTokens])`
③ When `earned()` returns 0 for any token due to precision loss, `lastEarnTime` is still advanced
④ Legitimate users permanently lose accumulated small rewards from prior epochs
⑤ Attacker can grief multiple users at minimal gas cost, causing systematic fund loss

Result: Small veNFT holders systematically lose earned rewards due to integer division truncation, creating unfair distribution favoring large holders.

Recommendation

We propose to fix it with any of these recommendations:

1. **Accumulate fractional amounts until claimable:**

```
mapping(address => mapping(uint256 => uint256)) public
    fractionalRewards;

function _calculateReward(...) internal returns (uint256) {
    uint256 fullAmount = (tokenRewardsPerEpoch * balance) / totalSupply
        ;
    uint256 remainder = (tokenRewardsPerEpoch * balance) % totalSupply;

    fractionalRewards[token][tokenId] += remainder;

    // Convert accumulated fractional to full units
    uint256 additionalReward = fractionalRewards[token][tokenId] /
        totalSupply;
    fractionalRewards[token][tokenId] %= totalSupply;

    return fullAmount + additionalReward;
}
```

2. **Use fixed-point arithmetic library:**

```
import "@prb/math/contracts/PRBMathUD60x18.sol";
using PRBMathUD60x18 for uint256;

// Perform calculations with 18 decimal precision
uint256 proportionalReward = tokenRewardsPerEpoch[token][_currTs]
    .mul(dustLock.balanceOfNFTAt(tokenId, _currTs).fromUint())
    .div(tokenSupplyBalanceCurrTs.fromUint())
    .toUint();
```

References

1. SCSVS G4: Business Logic

[NRL-6c19a5e-H03] Permanent reward loss for burnt veNFTs

HIGH FIXED

Retest (2025-08-14)

The vulnerability has been completely resolved. The team integrated `DustLock` with `RevenueReward` so that rewards are automatically claimed before any veNFT burn, transfer, or removal operation. When `_notifyAfterTokenBurned()` or `_notifyAfterTokenTransferred()` are called, `RevenueReward` automatically claims all pending rewards via `_claimRewardsTo()` and sends them to the previous owner, preventing permanent reward loss.

Affected files

- DustLock.sol#L755
- DustLock.sol#L782
- DustLock.sol#L832
- DustLock.sol#L878

Description

The DustLock contract allows veNFTs to be burned through various operations (`withdraw`, `earlyWithdraw`, `merge`, `split`) without checking if the veNFT has unclaimed rewards in the RevenueReward contract. When a veNFT is burned, its tokenId becomes invalid, making any accumulated rewards permanently inaccessible since the RevenueReward.getReward() function requires a valid tokenId to claim rewards.

The issue stems from the lack of integration between DustLock (which manages veNFT lifecycle) and RevenueReward (which tracks reward accruals). DustLock burns veNFTs without consulting RevenueReward about pending rewards, creating a permanent disconnect between reward calculations and claimability.

Vulnerable scenario

The following steps lead to the described result:

① User owns veNFT #123 and participates in the protocol for several weeks/months
② RevenueReward contract accumulates rewards for veNFT #123 based on the user's voting power and protocol revenue

③ User decides to exit their position and calls `withdraw(123)` or `earlyWithdraw(123)` without first claiming rewards
④ DustLock burns veNFT #123 without checking for unclaimed rewards in RevenueReward
⑤ User attempts to claim rewards via `getReward(123, tokens)` but the call fails because `dustLock.ownerOf(123)` reverts (NFT no longer exists)
⑥ All accumulated rewards for veNFT #123 become permanently inaccessible and remain stuck in the RevenueReward contract
⑦ The same issue occurs with merge/split operations where the original veNFT is burned

Result: Users permanently lose all unclaimed rewards when their veNFTs are burned, leading to protocol fund leakage and user financial loss.

Recommendation

1. **Add unclaimed reward validation before burning (Recommended):** Implement a check in all burn operations to ensure no rewards are pending:

```
function _requireNoUnclaimedRewards(uint256 _tokenId) internal view {
    if (revenueReward != address(0)) {
        address[] memory tokens = IRevenueReward(revenueReward).
            getRewardTokens();
        for (uint256 i = 0; i < tokens.length; i++) {
            uint256 earned = IRevenueReward(revenueReward).earned(tokens
                [i], _tokenId);
            if (earned > 0) {
                revert UnclaimedRewards(_tokenId, tokens[i], earned);
            }
        }
    }
}
```

2. **Auto-claim before burn (Alternative):** Automatically claim all rewards before burning the veNFT:

```
function _autoClaimRewards(uint256 _tokenId, address recipient)
    internal {
    if (revenueReward != address(0)) {
        address[] memory tokens = IRevenueReward(revenueReward).
            getRewardTokens();
        IRevenueReward(revenueReward).getReward(_tokenId, tokens);
    }
}
```

References

1. SCSVS G4: Business Logic

[NRL-6c19a5e-M01] Self-repaying loan reward receiver persists across veNFT transfers

MEDIUM FIXED

Retest (2025-08-14)

The vulnerability has been completely resolved. The team added `_notifyAfterTokenTransferred()` integration that automatically claims pending rewards for the previous owner and resets the self-repaying loan reward receiver to zero (`tokenRewardReceiver[_tokenId] = address(0)`) on veNFT transfers.

Affected files

- RevenueReward.sol#L53-L57
- RevenueReward.sol#L68-L71

Description

The self-repaying loan mechanism in RevenueReward allows veNFT owners to redirect their rewards to a specified address via `enableSelfRepayLoan()`. However, when a veNFT is transferred to a new owner, the reward receiver configuration persists unchanged. This creates a silent reward theft scenario where the new owner's rewards continue flowing to the previous owner's designated address until the new owner discovers the issue and manually calls `disableSelfRepayLoan()`.

The vulnerability stems from the lack of integration between DustLock (which handles veNFT transfers) and RevenueReward (which manages reward distribution). When `transferFrom()` is called on DustLock, it updates NFT ownership but does not notify RevenueReward to reset the reward receiver configuration.

Vulnerable scenario

The following steps lead to the described result:

① Alice owns veNFT #123 and calls `enableSelfRepayLoan(123, alice_address)` to redirect rewards to her address
② Alice transfers or sells veNFT #123 to Bob using `DustLock.transferFrom()`

③ Bob becomes the new owner but `tokenRewardReceiver[123]` still points to Alice's address
④ When rewards are distributed, `getReward()` sends all rewards to Alice instead of Bob
⑤ Bob experiences weeks/months of missing rewards before discovering the issue
⑥ Bob must manually call `disableSelfRepayLoan(123)` to reclaim future rewards
⑦ Bob has lost all rewards accumulated during the period of silent redirection

Result: New veNFT owners unknowingly lose rewards to previous owners until they discover and fix the configuration, causing financial loss and poor user experience.

Recommendation

Auto-reset on transfer: Modify DustLock to notify RevenueReward when NFTs are transferred, automatically resetting the reward receiver:

```
// In DustLock._transferFrom()
if (address(revenueReward) != address(0)) {
    IRevenueReward(revenueReward).resetRewardReceiver(_tokenId);
}
```

References

1. SCSVS G4: Business Logic

[NRL-6c19a5e-M02] Missing permanent lock validation in split function

MEDIUM FIXED

Retest (2025-08-14)

The vulnerability has been completely resolved. The team added explicit permanent lock validation (`if (newLocked.isPermanent)revert PermanentLock();`) in the split function before any operations proceed.

Affected files

- DustLock.sol#L861-L900

Description

The `split()` function in DustLock.sol fails to validate that the source veNFT is not a permanent lock before allowing the split operation. According to the interface documentation

in IDustLock.sol, the split function should "Cannot split permanent locks or locks that have already voted in the current epoch" (line 578). However, the implementation only checks for lock expiration, split permissions, and ownership, but completely omits the permanent lock validation.

This allows users to split permanent locks into multiple smaller permanent locks, each retaining the permanent status and maximum voting power characteristics.

Vulnerable scenario

The following steps lead to the described result:

① The user creates a permanent lock with a large amount of DUST tokens (e.g., 1,000,000 DUST).
② The user calls `split()` function with their permanent lock tokenId and specifies an amount (e.g., 100,000 DUST).
③ The contract successfully splits the permanent lock into two new permanent locks without reverting.
④ The user now has two permanent locks (900,000 DUST and 100,000 DUST) instead of one indivisible permanent lock.
⑤ The user can distribute these fragmented permanent locks to different addresses for coordinated voting or sell portions while maintaining permanent voting power benefits.

Result: Users can fragment permanent locks to manipulate voting power distribution and undermine the protocol's permanent commitment mechanism.

Recommendation

Add explicit permanent lock validation in the `split()` function before proceeding with the split operation:

```
function split(uint256 _from, uint256 _amount) external nonReentrant returns (
    uint256 _tokenId1, uint256 _tokenId2) {
    address sender = _msgSender();
    address owner = _ownerOf(_from);
    if (owner == address(0)) revert SplitNoOwner();
    if (!canSplit[owner] && !canSplit[address(0)]) revert SplitNotAllowed();
    if (!_isApprovedOrOwner(sender, _from)) revert NotApprovedOrOwner();
    LockedBalance memory newLocked = _locked[_from];

    // Add missing permanent lock validation
    if (newLocked.isPermanent) revert PermanentLock();

    if (newLocked.end <= block.timestamp && !newLocked.isPermanent) revert
```

```
        LockExpired();
    // ... rest of function
}
```

References

1. SCSVS G4: Business Logic

[NRL-6c19a5e-M03] Inability to delegate claim permissions by users

MEDIUM FIXED

Retest (2025-08-14)

The vulnerability has been completely resolved. The team updated the access control in `setClaimer()` to allow veNFT owners to delegate claim permissions for their own accounts (`if (msg.sender != user)check`), while preserving admin override capabilities.

Affected files

- DustRewardsController.sol#L167-L171

Description

The `setClaimer` function in DustRewardsController has an access control mismatch between its implementation and interface documentation. The interface explicitly states that this function should be "Only callable by users for their own accounts or by admin" (IDustRewardsController.sol L51). However, the implementation restricts access to only the emissionManager role:

```
function setClaimer(address user, address caller) external override
    onlyEmissionManager {
    _authorizedClaimers[user] = caller;
    emit ClaimerSet(user, caller);
}
```

This discrepancy prevents users from delegating claim permissions for their own rewards, which contradicts the stated intent in the interface and limits user autonomy in the protocol.

Vulnerable scenario

The following steps lead to the described result:

① A user, who is accruing or has accrued rewards within the Neverland protocol, wishes to delegate the authority to claim these rewards to another specific address (referred to as a claimer). This delegation is intended to facilitate integration with other protocols or enable trusted services to manage reward claims on their behalf, as outlined by the IDustRewardsController interface.

① The user attempts to call the setClaimer function on the DustRewardsController contract, providing their own address as the user parameter and the desired delegate address as the claimer parameter. This action aligns with the IDustRewardsController interface's specification that the function is "Only callable by users for their own accounts or by admin.

① However, the setClaimer function in the DustRewardsController.sol implementation is protected by the onlyEmissionManager modifier. Consequently, the user's transaction to set their own claimer is reverted, as only the designated emissionManager (administrator) address is permitted to execute this function.

Result: Users cannot authorize third parties to claim rewards on their behalf, requiring administrative intervention for all claimer delegations and reducing protocol flexibility.

Recommendation

Implement one of the following solutions to align the implementation with the interface documentation:

1. Allow both users and admin to call the function by modifying the access control:

```
function setClaimer(address user, address caller) external override {
    // Allow users to set claimers for their own accounts
    if (msg.sender != user) {
        // If not the user themselves, require admin permission
        require(msg.sender == emissionManager, "
            ONLY_EMISSION_MANAGER_OR_SELF");
    }
    _authorizedClaimers[user] = caller;
    emit ClaimerSet(user, caller);
}
```

2. If the intent is to restrict this functionality to admin only, update the interface documentation to accurately reflect this limitation:

```
/**
 * @notice Authorizes an address to claim rewards on behalf of another
```

```
    user
* @dev Establishes a delegation relationship for reward claiming
*      This is useful for integrating with other protocols or allowing
*      trusted services to manage reward claims for users
*      Only callable by the emission manager
* ...
*/
```

References

1. SCSVS G5: Access control

[NRL-6c19a5e-L01] One-step ownership transfer in setTeam

LOW FIXED

Retest (2025-08-14)

The team replaced the direct one-step `setTeam()` function with a comprehensive two-step ownership transfer pattern: `proposeTeam()-> acceptTeam()`.

Affected files

- DustLock.sol

Description

The `setTeam` function transfers ownership immediately without a two-step confirmation process, risking permanent loss of admin control in case of a mistake.

Result: Accidental transfers to incorrect addresses are irreversible and could result in permanent loss of admin control over the protocol.

Recommendation

Implement a two-step ownership transfer pattern with pending owner acceptance to prevent accidental transfers.

References

1. SCSVS G5: Access control

[NRL-6c19a5e-L02] DustLock NFT can be transferred to address 0

LOW FIXED

Retest (2025-08-14)

The team added explicit zero address validation (`CommonChecksLibrary.revertIfZeroAddress(_to)`) at the beginning of the `_transferFrom` function.

Affected files

- DustLock.sol

Description

The `_transferFrom` function in `DustLock.sol` lacks validation to prevent transfers to the zero address, which can lead to NFTs being permanently locked with their DUST balance. While this does not lead to any significant impact, it may be an overlook, as it allows to create "dead" NFTs instead of burning them. As a result of such transfer, address 0 may accumulate locked NFTs.

```
// DustLock.sol
function _transferFrom(address _from, address _to, uint256 _tokenId, address
    _sender) internal {
    // Missing check: require(_to != address(0), "Transfer to zero address");

    if (!_isApprovedOrOwner(_sender, _tokenId)) revert NotApprovedOrOwner();
    if (_ownerOf(_tokenId) != _from) revert NotOwner();
    delete idToApprovals[_tokenId];

    _removeTokenFrom(_from, _tokenId);
    _addTokenTo(_to, _tokenId); // Will set owner to address(0)
}
```

Result of the attack: Unexpected protocol state.

Recommendation

Add a require statement in `_transferFrom` to check that `_to != address(0)`.

References

1. SCSVS G4: Business logic

[NRL-6c19a5e-L03] Non-compliant ERC721 totalSupply implementation

LOW ACKNOWLEDGED

Retest (2025-08-14)

After further investigation it was determined that the ERC721 standard does not require a `totalSupply()` function - this is only required by the `ERC721Enumerable` extension, which `DustLock` does not implement.

Affected files

- DustLock.sol#L988-L990

Description

The `DustLock` contract implements a `totalSupply()` function that returns the aggregate voting power of all veNFTs instead of the total number of existing NFTs, violating ERC721 expectations:

```
function totalSupply() external view returns (uint256) {
    return _supplyAt(block.timestamp); // Returns total voting power, not NFT
        count
}
```

The ERC721 standard expects `totalSupply()` to return the count of existing tokens (NFTs), but DustLock returns the sum of all voting power across all veNFTs. This creates a fundamental semantic mismatch that breaks compatibility with third-party integrations.

Result: Third-party integrations (NFT marketplaces, wallets, analytics tools, indexers) will display incorrect information and may malfunction when interacting with DustLock NFTs.

Recommendation

Add a separate NFT count function to provide true ERC721-compliant total supply:

```
// Track actual NFT count
uint256 private _totalNFTCount;
```

```
// ERC721-compliant total supply
function totalSupply() external view returns (uint256) {
    return _totalNFTCount;
}

// Separate function for voting power
function totalVotingPower() external view returns (uint256) {
    return _supplyAt(block.timestamp);
}

// Update count on mint/burn
function _mint(address to, uint256 tokenId) internal {
    // ... existing mint logic ...
    _totalNFTCount++;
}

function _burn(uint256 tokenId) internal {
    // ... existing burn logic ...
    _totalNFTCount--;
}
```

References

1. SCSVS C1: Token

[NRL-6c19a5e-L04] Single-step ownership transfer in upgradeable contract

LOW FIXED

Retest (2025-08-14)

The team upgraded `Dust.sol` from `OwnableUpgradeable` to `Ownable2StepUpgradeable`, implementing OpenZeppelin's standard two-step ownership transfer pattern.

Affected files

- Dust.sol#L16

Description

The Dust.sol contract uses `OwnableUpgradeable`, which implements single-step ownership transfer. This means when `transferOwnership(newOwner)` is called, ownership is immediately transferred to the specified address without requiring confirmation from the new owner.

If the new owner's address is incorrect, the ownership will be permanently transferred to an inaccessible address, resulting in complete loss of admin control over the contract.

Since Dust.sol is an upgradeable contract with critical administrative functions including `pause()`, `unpause()`, and upgrade capabilities, losing ownership would render these functions permanently inaccessible and could compromise the entire protocol.

Result: Permanent loss of admin control over critical protocol functions.

Recommendation

Replace `OwnableUpgradeable` with `Ownable2StepUpgradeable` to implement a two-step ownership transfer process:

1. **Import change:**

```
import {Ownable2StepUpgradeable} from "@openzeppelin/contracts-
    upgradeable/access/Ownable2StepUpgradeable.sol";
```

2. **Contract inheritance change:**

```
contract Dust is
    Initializable,
    ERC20Upgradeable,
    ERC20PausableUpgradeable,
    Ownable2StepUpgradeable, // Changed from OwnableUpgradeable
    ERC20PermitUpgradeable
```

References

1. SCSVS G5: Access control

[NRL-6c19a5e-L05] Missing minimum transaction amount validation enables network congestion through dust attacks

LOW FIXED

Retest (2025-08-20)

The team implemented configurable minimum lock amount validation (`minLockAmount = 1e18`) across all critical functions (`_createLock`, `_increaseAmountFor`, and `split`). The `split` function was verified on a separate commit f3d53ad8b0489d564120a8601f3fb9f6d92de487.

Affected files

- DustLock.sol#L681
- DustLock.sol#L710
- DustLock.sol#L874

Description

The DustLock contract only validates against zero amounts but does not enforce meaningful minimum transaction thresholds. Functions like `createLock`, `increaseAmount`, and `split` accept any amount above zero, including economically insignificant values as small as 1 wei.

```
// In _createLock
if (_value == 0) revert ZeroAmount(); // Only prevents zero, not dust amounts

// In _increaseAmountFor
if (_value == 0) revert ZeroAmount(); // Same issue

// In split
if (_splitAmount == 0) revert ZeroAmount(); // Same issue
```

An attacker can exploit this by creating numerous micro-transactions:

- `createLock(1, MINTIME)` - Creates 1 wei veNFT lock
- `increaseAmount(tokenId, 1)` - Adds 1 wei to existing lock
- `split(tokenId, 1)` - Splits off 1 wei amounts

Each transaction consumes network resources, creates storage overhead (new NFTs, checkpoints, user mappings), and forces other users to pay higher gas fees, while costing the attacker minimal amounts (1 wei + gas per transaction).

Result: The protocol is vulnerable to dust attacks that can congest the network and create storage bloat at minimal cost to attackers.

Recommendation

Implement meaningful minimum transaction amounts to prevent dust attacks:

```
uint256 public constant MIN_LOCK_AMOUNT = 1e18; // 1 DUST minimum

function _createLock(uint256 _value, uint256 _lockDuration, address _to)
    internal returns (uint256) {
    if (_value == 0) revert ZeroAmount();
    if (_value < MIN_LOCK_AMOUNT) revert AmountTooSmall(); // Add this check
    // ... rest of function
}
```

Apply similar minimum amount checks to:

- `_increaseAmountFor` function
- `split` function
- Any other functions accepting token amounts

Consider setting the minimum to a reasonable economic threshold (e.g., 1 DUST token) to prevent spam while maintaining accessibility for legitimate users.

References

1. SCSVS G8: Denial of Service

[NRL-6c19a5e-L06] Missing ownership validation in reward claiming leads to accidental token loss

LOW FIXED

Retest (2025-08-14)

The team added explicit ownership validation (`if (owner != to)revert NotTokenOwner()`) in the `DustLockTransferStrategy.performTransfer()` function before depositing claimed rewards into existing veNFTs.

Affected files

- DustRewardsController.sol#L206-L239
- DustLockTransferStrategy.sol#L60-L64

Description

The reward claiming system allows users to specify any `tokenId` when claiming their earned rewards, without validating that the user owns the specified veNFT. This creates a risk of accidental token loss when users mistakenly specify an incorrect `tokenId`.

When users call `claimRewards()` or related functions, they can provide any `tokenId` parameter:

```
function claimRewards(
    address[] calldata assets,
    uint256 amount,
    address to,
    address reward,
    uint256 lockTime,
    uint256 tokenId // No ownership validation
) external returns (uint256)
```

The system validates that the user has legitimate rewards to claim, but does not verify that the user owns the specified `tokenId`. In `DustLockTransferStrategy.performTransfer()`, the only check performed is:

```
if (DUST_LOCK.ownerOf(tokenId) == address(0)) revert InvalidTokenId();
```

This only ensures the NFT exists, not that the claimer owns it. Consequently, a user's earned rewards can be deposited into any valid veNFT, including those owned by other users.

Result: Users can accidentally lose their earned rewards by specifying an incorrect `tokenId`, with no way to recover the tokens once deposited into another user's veNFT.

Recommendation

Add ownership validation in the reward claiming process to prevent accidental token loss:

Add ownership check in DustLockTransferStrategy:

```
if (tokenId > 0) {
    if (DUST_LOCK.ownerOf(tokenId) == address(0)) revert InvalidTokenId();
    if (DUST_LOCK.ownerOf(tokenId) != to) revert NotTokenOwner(); // Add
        this check
    IERC20(reward).approve(address(DUST_LOCK), amount);
    DUST_LOCK.depositFor(tokenId, amount);
}
```

References

1. SCSVS G4: Business Logic

[NRL-6c19a5e-L07] Precision loss in voting power calculation due to division before multiplication

LOW FIXED

Retest (2025-08-14)

The team adopted PRB-Math UD60x18 for absolute precision in voting power calculations, replacing integer division with fixed-point arithmetic that maintains 18-decimal precision.

Affected files

- DustLock.sol#L461-L468

Description

In the `_checkpoint` function, there's a potential precision loss issue in the calculation of slope and bias values due to performing division before multiplication:

```
if (_oldLocked.end > block.timestamp && _oldLocked.amount > 0) {
    uOld.slope = _oldLocked.amount / iMAXTIME;
    uOld.bias = uOld.slope * (_oldLocked.end - block.timestamp).toInt128();
}
if (_newLocked.end > block.timestamp && _newLocked.amount > 0) {
    uNew.slope = _newLocked.amount / iMAXTIME;
    uNew.bias = uNew.slope * (_newLocked.end - block.timestamp).toInt128();
}
```

The calculation first divides `_oldLocked.amount` or `_newLocked.amount` by `iMAXTIME` to get the slope, and then multiplies by the time difference to get the bias. This approach can result in precision loss due to integer division truncation, especially for small amounts.

Result: The voting power (bias) calculation may be less accurate than intended, potentially resulting in slightly lower voting power than users should have. The impact increases with smaller token amounts, where the division by `iMAXTIME` (which is a large number representing approximately 1 year in seconds) could result in significant truncation.

Recommendation

We propose two potential approaches:

- Option 1: Accept the current implementation has precision loss and mention it in the documentation.
- Option 2: Use Fixed-Point Math Library. If higher precision is required, implement a fixed-point math library such as Solady's `FixedPointMathLib`:

```
import {FixedPointMathLib} from "solady/utils/FixedPointMathLib.sol";

// Update Point struct to use fixed-point values
struct Point {
    int256 bias; // Now stores WAD values (18 decimals)
    int256 slope; // Now stores WAD values (18 decimals)
    uint256 ts;
    uint256 blk;
}

// Updated calculation with consistent fixed-point precision
if (_oldLocked.end > block.timestamp && _oldLocked.amount > 0) {
    uint256 slopeWad = FixedPointMathLib.divWad(_oldLocked.amount, uint256(
        iMAXTIME));
    uint256 biasWad = FixedPointMathLib.mulWad(slopeWad, uint256(_oldLocked.end
         - block.timestamp));
    uOld.slope = int256(slopeWad);
    uOld.bias = int256(biasWad);
}
```

References

1. SCSVS G7: Arithmetic

[NRL-6c19a5e-L08] Inability to collect any rewards

LOW FIXED

Retest (2025-08-14)

The vulnerability has been completely resolved. The team implemented `_notifyTokenMinted` to initialize `tokenMintTime` on lock creation, preventing expensive historical reward loops from contract deployment. They also added `getRewardUntilTs` for partial-period claims, reducing gas costs in long-unclaimed scenarios.

Note: Follow-up testing confirmed that the originally reported 700M gas case was only reproducible under extreme, artificial time-warp scenarios. In realistic conditions with weekly checkpoints, the maximum observed cost was about 4M gas after 5 years without claims, well under Monad's 150M gas limit. The severity was therefore downgraded to low.

Affected files

- RevenueReward.sol#L117-L117

Description

The `lastEarnTime` storage variable is used to store the timestamp of when there was latest claim for particular `tokenId`. It is used in the `earned` function to calculate `_startTs`, the starting timestamp of the reward collection period.

The issue is that `lastEarnTime` is never initialised so it's default value is 0. Therefore, when the user calls the `getReward` function for the first time, the difference between `_endTs` and `_startTs` is equal to the current timestamp, meaning that the loop will iterate over 2900 weeks and cost at least 700M gas.

There is also an issue with the same result when the user does not get a reward for a long time (e.g. 5 years) the period can also grow and exceed the 30M gas limit.

Vulnerable scenario

The following steps lead to the described result:

① The user creates a lock.
② The revenue rewards are sent to the RevenueReward contract.
③ User tries to call `getReward` but it always reverts.

Result: Permanent lock of all revenue rewards.

Recommendation

Store the timestamp of contract's deployment and get the `lastEarnTime` value using the following formula: `uint256 lastEarnTime = Math.max(lastEarnTime[token][tokenId], deployedTimestamp);`
Add the `endTimestamp` parameter to the `getReward` function that allows the user to limit the size of the time period. The parameter has to be validated to be smaller than `block.timestamp`.

References

1. SCSVS G4: Business logic

6. Recommendations

[NRL-6c19a5e-R01] Do not use assert

INFO IMPLEMENTED

Retest (2025-08-14)

The team eliminated all assert/require statements from the codebase and replaced them with proper error handling using custom errors.

Description

Multiple functions use `assert` statements which consume all remaining gas on failure instead of using require statements that refund unused gas. Failed transactions consume all gas instead of refunding unused gas, resulting in unnecessary costs for users. This decreases user experience and leads to increased gas costs for users.

Recommendation

Replace `assert` statements with `require` statements to enable gas refunds on reverts.

References

1. SCSVS G10: Gas usage & limitations

[NRL-6c19a5e-R02] Consider using newest Solidity version

INFO ACKNOWLEDGED

Retest (2025-08-14)

The team strategically chose to maintain consistent use of Solidity 0.8.19 in all contracts rather than upgrading to 0.8.31

Description

In accordance with the best security practices, it is recommended to use the latest stable versions of major Solidity releases.

Very often, older versions contain bugs that have been discovered and fixed in newer versions. Moreover, it is worth remembering that the version should be clearly specified so that

all tests and compilations are performed with the same version.

The current implementation uses:

- Most contracts use `pragma solidity 0.8.19;` (fixed version)
- IDustLock.sol uses `pragma solidity ^0.8.0;` (flexible version)
- DustRewardsController.sol uses `pragma solidity ^0.8.10;` (flexible version)

Recommendation

Use the latest stable version of major Solidity release:

```
pragma solidity 0.8.31;
```

Note: If it is planned to deploy on multiple chains, stay aware that some of them don't support `PUSH0` opcode. If solc `>=0.8.20` is used, the `PUSH0` opcode will be present in the bytecode. In this situation, it is recommended to choose `0.8.19`.

References

1. SCSVS G1: Architecture, design and threat modeling
2. Floating pragma

[NRL-6c19a5e-R03] Consider validating the voting state for locks before splitting and merging

INFO ACKNOWLEDGED

Retest (2025-08-14)

The team did not add explicit voting state validation before splitting and merging operations. The current implementation includes standard validations (ownership, approval, lock expiry, permanent lock checks), but does not specifically validate voting participation state as suggested in the recommendation.

Description

The `merge()` and `split()` functions in DustLock.sol fail to validate that the source and destination veNFTs have not already voted in the current epoch before allowing the merge or split operation. According to the interface documentation in IDustLock.sol:

- "Cannot merge source veNFTs that are permanent or have voted in the current epoch" (line 598),

- "Cannot split permanent locks or locks that have already voted in the current epoch" (line 578).

However, the implementation completely omits any validation of the voting state and does not utilize the defined `AlreadyVoted()` error.

The functions are missing the critical voting state validation that would prevent users from merging veNFTs after they have already participated in governance voting. This creates a loophole where users can effectively consolidate multiple votes by merging voted veNFTs and then using the merged result to vote again in the same epoch.

Note: The severity of this finding has been downgraded to a recommendation. The project team clarified that their governance process is conducted off-chain, utilizing periodic checkpoints for vote calculation. This architecture mitigates the described attack vector, which is only feasible in an on-chain environment. This issue remains documented to ensure the team is aware of the potential vulnerability should they migrate their voting system on-chain in the future.

The following steps demonstrate voting power multiplication through merge:

1. The user creates two veNFT locks with significant amounts of DUST tokens (e.g., 300,000 and 200,000 DUST).
2. A critical governance proposal is active, and the user votes with both veNFTs, casting votes based on their respective locked token amounts (total: 500,000 DUST voting power).
3. Immediately after voting with both veNFTs, the user calls the `merge()` function to combine the two voted veNFTs into a single veNFT (500,000 DUST).
4. The contract successfully merges the veNFTs without checking if either has already voted in the current epoch.
5. The user now has a single merged veNFT that can vote again on the same proposal, effectively adding another 500,000 DUST voting power to their original votes.
6. The user votes with the merged veNFT, manipulating the governance outcome by casting additional votes in the same epoch with the same underlying economic stake.
7. **This process can be repeated indefinitely** - the user can continue merging voted veNFTs and voting with the merged results to achieve unlimited voting power multiplication within a single epoch.

Recommendation

Add explicit voting state validation in the `merge()` function before proceeding with the merge operation. This requires implementing a mechanism to track voting participation per epoch and checking this state for both source and destination veNFTs:

```
function merge(uint256 _from, uint256 _to) external nonReentrant {
```

```
    address sender = _msgSender();
    if (_from == _to) revert SameNFT();
    if (!_isApprovedOrOwner(sender, _from)) revert NotApprovedOrOwner();
    if (!_isApprovedOrOwner(sender, _to)) revert NotApprovedOrOwner();

    // Add missing voting state validation for both veNFTs
    if (hasVotedInCurrentEpoch(_from)) revert AlreadyVoted();
    if (hasVotedInCurrentEpoch(_to)) revert AlreadyVoted();

    LockedBalance memory oldLockedTo = _locked[_to];
    if (oldLockedTo.end <= block.timestamp && !oldLockedTo.isPermanent) revert
        LockExpired();
    // ... rest of function
}
```

Add explicit voting state validation in the `split()` function before proceeding with the split operation. This requires implementing a mechanism to track voting participation per epoch and checking this state:

```
function split(uint256 _from, uint256 _amount) external nonReentrant returns (
    uint256 _tokenId1, uint256 _tokenId2) {
    address sender = _msgSender();
    address owner = _ownerOf(_from);
    if (owner == address(0)) revert SplitNoOwner();
    if (!canSplit[owner] && !canSplit[address(0)]) revert SplitNotAllowed();
    if (!_isApprovedOrOwner(sender, _from)) revert NotApprovedOrOwner();
    LockedBalance memory newLocked = _locked[_from];

    // Add missing voting state validation
    if (hasVotedInCurrentEpoch(_from)) revert AlreadyVoted();

    if (newLocked.isPermanent) revert PermanentLock();
    if (newLocked.end <= block.timestamp && !newLocked.isPermanent) revert
        LockExpired();
    // ... rest of function
}
```

Additionally, implement the `hasVotedInCurrentEpoch()` function to track and validate voting participation per epoch, ensuring that veNFTs that have already voted cannot be merged until the next epoch begins.

References

1. SCSVS G4: Business Logic

[NRL-6c19a5e-R04] Use named constants instead of magic numbers

INFO IMPLEMENTED

Retest (2025-08-14)

The team successfully defined named constants for all magic numbers to improve code readability and maintainability. The implementation includes `BASIS_POINTS = 10_000`, `MAX_USER_POINTS = 1_000_000_000`, and `MAX_CHECKPOINT_ITERATIONS = 255` as internal constants, replacing the hardcoded values throughout the codebase.

Description

The Neverland protocol contains several hardcoded magic numbers throughout the codebase without defining them as named constants. This practice reduces code readability, maintainability, and increases the risk of errors during future modifications.

Specific instances found:

1. **Basis Points Divisor (10_000):**

```
// DustLock.sol:788
uint256 userPenaltyAmount = earlyWithdrawPenalty * balanceOfNFT(_tokenId) /
    10_000;

// DustLockTransferStrategy.sol:71
uint256 treasuryValue = (amount * DUST_LOCK.earlyWithdrawPenalty()) / 10_000
    ;

// DustLock.sol:807
if (_earlyWithdrawPenalty >= 10_000) revert InvalidWithdrawPenalty();
```

2. **Array Size Limit (1_000_000_000):**

```
// DustLock.sol:407
mapping(uint256 => UserPoint[1000000000]) internal _userPointHistory;

// BalanceLogicLibrary.sol:24, 98
mapping(uint256 => IDustLock.UserPoint[1000000000]) storage
```

```
    _userPointHistory,
```

3. **Loop Iteration Limit (255):**

```
// DustLock.sol:509
for (uint256 i = 0; i < 255; ++i) {

// BalanceLogicLibrary.sol:139
for (uint256 i = 0; i < 255; ++i) {
```

Recommendation

Define named constants for all magic numbers to improve code readability and maintainability:

```
// Add to contract or library
uint256 private constant BASIS_POINTS = 10_000;
uint256 private constant MAX_USER_POINTS = 1_000_000_000;
uint256 private constant MAX_CHECKPOINT_ITERATIONS = 255;

// Usage examples:
uint256 userPenaltyAmount = earlyWithdrawPenalty * balanceOfNFT(_tokenId) /
    BASIS_POINTS;
mapping(uint256 => UserPoint[MAX_USER_POINTS]) internal _userPointHistory;
for (uint256 i = 0; i < MAX_CHECKPOINT_ITERATIONS; ++i) {
```

References

1. SCSVS G1: Architecture, design and threat modeling

[NRL-6c19a5e-R05] Make error message formatting consistent

INFO IMPLEMENTED

Retest (2025-08-14)

The team successfully standardized on custom errors throughout the protocol for consistency and gas efficiency. The implementation includes comprehensive custom error definitions in interfaces, eliminating string-based require statements in favor of gas-efficient custom errors with descriptive names.

Description

The Neverland protocol uses inconsistent error handling patterns throughout the codebase, mixing string-based `require` statements with custom error types. This inconsistency creates confusion for developers and results in higher gas costs for string-based errors.

Specific instances found:

1. **String-based errors in DustRewardsController:**

```
require(user != address(0), "INVALID_USER_ADDRESS");
require(to != address(0), "INVALID_TO_ADDRESS");
require(address(transferStrategy) != address(0), "STRATEGY_CAN_NOT_BE_ZERO")
    ;
require(_isContract(address(transferStrategy)) == true, "
    STRATEGY_MUST_BE_CONTRACT");
```

2. **Custom errors in DustLock (better practice):**

```
if (_value == 0) revert ZeroAmount();
if (unlockTime <= block.timestamp) revert LockDurationNotInFuture();
if (unlockTime > block.timestamp + MAXTIME) revert LockDurationTooLong();
```

Recommendation

Standardize on custom errors throughout the protocol for consistency and gas efficiency:

```
// Define custom errors
error InvalidUserAddress();
error InvalidToAddress();
error StrategyCannotBeZero();
error StrategyMustBeContract();

// Replace require statements
if (user == address(0)) revert InvalidUserAddress();
if (to == address(0)) revert InvalidToAddress();
if (address(transferStrategy) == address(0)) revert StrategyCannotBeZero();
if (!_isContract(address(transferStrategy))) revert StrategyMustBeContract();
```

References

1. SCSVS G1: Architecture, design and threat modeling

[NRL-6c19a5e-R06] Add zero-address validation in critical functions

INFO IMPLEMENTED

Retest (2025-08-14)

The team successfully added comprehensive zero-address validation to all critical functions throughout the protocol. The implementation includes zero-address checks in constructors, administrative functions, and transfer operations.

Description

Several critical functions in the Neverland protocol lack proper zero-address validation for important parameters. This can lead to operational issues, loss of administrative control, or unexpected contract behavior when zero addresses are accidentally provided.

Specific instances found:

1. **RevenueReward.setRewardDistributor():**

```
function setRewardDistributor(address newRewardDistributor) external {
    if (_msgSender() != rewardDistributor) revert NotRewardDistributor();
    rewardDistributor = newRewardDistributor;
    // Missing: zero-address validation
}
```

2. **DustLock constructor and initialization functions:**

```
// Various constructor parameters lack zero-address checks
// Could result in contracts being deployed with invalid addresses
```

3. **Transfer strategy installation:**

```
function _installTransferStrategy(address reward, IDustTransferStrategy
    transferStrategy) internal {
    require(address(transferStrategy) != address(0), "
        STRATEGY_CAN_NOT_BE_ZERO");
    // Good: Has zero-address check
    // But reward parameter lacks validation
}
```

Recommendation

Add comprehensive zero-address validation to all critical functions:

```
// Example for RevenueReward.setRewardDistributor()
function setRewardDistributor(address newRewardDistributor) external {
    if (_msgSender() != rewardDistributor) revert NotRewardDistributor();
    if (newRewardDistributor == address(0)) revert InvalidAddress();
    rewardDistributor = newRewardDistributor;
}

// Example for transfer strategy installation
function _installTransferStrategy(address reward, IDustTransferStrategy
    transferStrategy) internal {
    if (reward == address(0)) revert InvalidRewardAddress();
    if (address(transferStrategy) == address(0)) revert InvalidTransferStrategy
        ();
    // ... rest of function
}

// Add custom errors
error InvalidAddress();
error InvalidRewardAddress();
error InvalidTransferStrategy();
```

Additional recommendations:

- Validate all address parameters in constructors and initialization functions
- Consider using a modifier for common zero-address checks
- Document which functions intentionally allow zero addresses (if any)

References

1. SCSVS G1: Architecture, design and threat modeling

[NRL-6c19a5e-R07] Remove redundant decimals

INFO IMPLEMENTED

Retest (2025-08-14)

The team successfully removed the redundant decimals constant from `DustLock.sol`. The contract now correctly implements only ERC721 metadata (name, symbol, version) without the unnecessary ERC20-style decimals constant, improving code clarity and

eliminating confusion about the contract's purpose as an NFT contract rather than a token contract.

Description

DustLock.sol declares a `decimals` constant which is not part of the ERC721 standard and serves no purpose in an NFT contract.

Recommendation

Remove the unused constant.

References

1. SCSVS G11: Code clarity

[NRL-6c19a5e-R08] Verify zero-address rewards receiver in RevenueReward

INFO IMPLEMENTED

Retest (2025-08-14)

The team has not added the recommended zero-address validation to the `getReward` function in `RevenueReward.sol`. The function still lacks the explicit check for `rewardsReceiver` being non-zero after calling `_resolveRewardsReceiver(tokenId)`.

Description

The `getReward` function in RevenueReward.sol doesn't verify that `rewardsReceiver` is non-zero after querying the NFT owner, potentially attempting transfers to address(0). Since the protocol allows transfers of NFTs to address(0), this is a reachable protocol state.

Recommendation

Function may attempt transfers to address(0) which should not be possible as it generates unnecessarily locked funds.

References

1. SCSVS G4: Business logic

[NRL-6c19a5e-R09] Fix value emitted in event

INFO IMPLEMENTED

Retest (2025-08-14)

The function now correctly emits `RecoverTokens(token, unnotifiedTokenAmount)` on line 322, providing the actual amount of tokens recovered rather than the total contract balance.

Description

In RevenueReward.sol, The `recoverTokens` function emits the total contract balance in the event instead of the actual amount of tokens recovered. Incorrect event data may cause confusion in off-chain systems and make it difficult to track actual recovered amounts.

Recommendation

Change `emit RecoverTokens(token, balance);` to `emit RecoverTokens(token, unnotifiedTokenAmount);`.

References

1. SCSVS G9: Blockchain data

[NRL-6c19a5e-R10] Require minimum lock amount

INFO ACKNOWLEDGED

Retest (2025-08-14)

The `earlyWithdraw()` function in `DustLock.sol` does not include the validation of the minimum lock amount. The function allows for an early withdrawal of any amount without checking if the locked amount meets the `minLockAmount` requirement.

Description

In DustLock.sol, function `earlyWithdraw`, the penalty calculation can round down to zero due to integer division by `10,000` when locks have small balance amounts associated with them.

While this does not cause significant security impact, this might be undesired from protocol design standpoint, breaking an invariant that a penalty has to be paid on early withdraw.

Recommendation

Require a minimum lock amount in `createLock` to ensure penalties are always meaningful, or implement a minimum penalty amount.

References

1. SCSVS G4: Business logic

[NRL-6c19a5e-R11] Fix naming convention

INFO IMPLEMENTED

Retest (2025-08-20)

The team successfully renamed the internal function from `earned` to `_earned` (line 253), following the proper naming convention in which internal and private functions should start with an underscore.
The `_notify`... functions' names were changed (removed underscore) in a separate commit b93f34a5cbe2251397068e4546ab870d1527bf3c.

Description

In RevenueReward.sol, the `earned` function is internal but its name does not start with an underscore, which may lead to confusion and decreases maintainability. The names of internal and private functions, according to best practices, should start with an underscore.

Recommendation

Change the function name to `_earned`.

References

1. SCSVS: G11 Code clarity

7. Impact on risk classification

Risk classification is based on the one developed by OWASP [1], however it has been adapted to the immutable and transparent code nature of smart contracts. The Web3 ecosystem forgives much less mistakes than in the case of traditional applications, the servers of which can be covered by many layers of security.

Therefore, the classification is more strict and indicates higher priorities for paying attention to security.

OVERALL RISK SEVERITY				
Impact on risk	HIGH	CRITICAL	HIGH	MEDIUM
	MEDIUM	MEDIUM	MEDIUM	LOW
	LOW	LOW	LOW	INFO
		HIGH	MEDIUM	LOW
		Likelihood		

[1] OWASP Risk Rating methodology

8. Long-term best practices

8.1. Use automated tools to scan your code regularly

It's a good idea to incorporate automated tools (e.g. slither) into the code writing process. This will allow basic security issues to be detected and addressed at a very early stage.

8.2. Perform threat modeling

Before implementing or introducing changes to smart contracts, perform threat modeling and think with your team about what can go wrong. Set potential targets of the attacker and possible ways to achieve them, keep it in mind during implementation to prevent bad design decisions.

8.3. Use Smart Contract Security Verification Standard

Use proven standards to maintain a high level of security for your contracts. Treat individual categories as checklists to verify the security of individual components. Expand your unit tests with selected checks from the list to be sure when introducing changes that they did not affect the security of the project.

8.4. Discuss audit reports and learn from them

The best guarantee of security is the constant development of team knowledge. To use the audit as effectively as possible, make sure that everyone in the team understands the mistakes made. Consider whether the detected vulnerabilities may exist in other places, audits always have a limited time and the developers know the code best.

8.5. Monitor your and similar contracts

Use the tools available on the market to monitor key contracts (e.g. the ones where user's tokens are kept). If you have used code from another project, monitor their contracts as well and introduce procedures to capture information about detected vulnerabilities in their code.

Damian Rusinek

Smart Contracts Auditor

@drdr_zz

damian.rusinek@composable-security.com

Paweł Kuryłowicz

Smart Contracts Auditor

@wh01s7

pawel.kurylowicz@composable-security.com