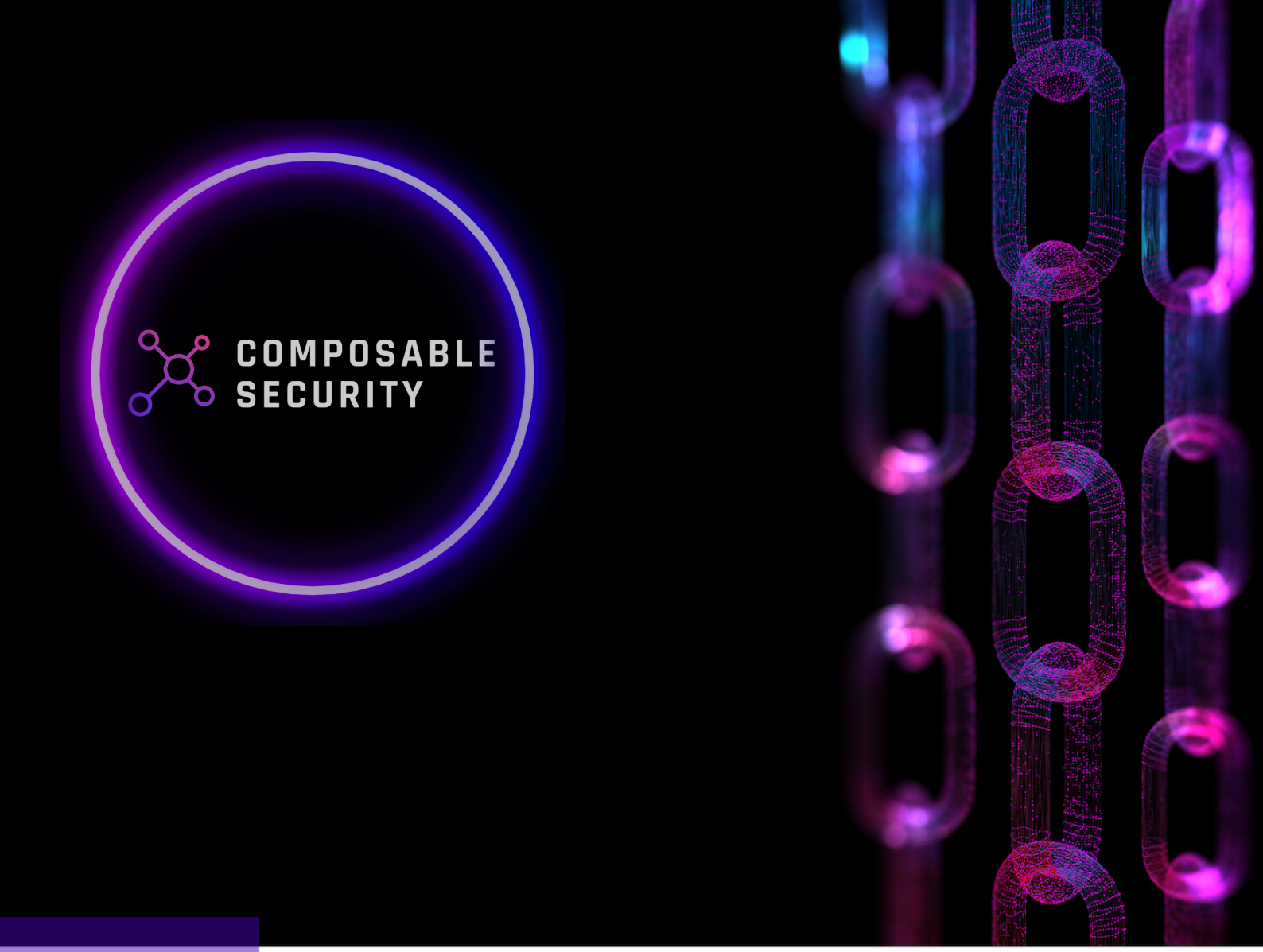


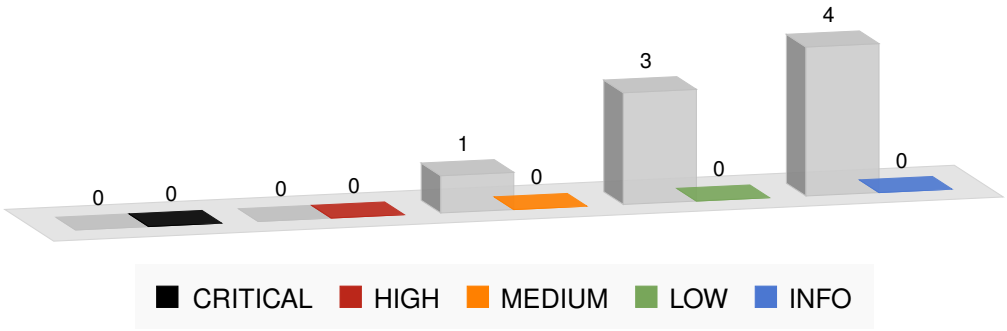
REPORT

Smart contract security review for Neverland Money

Prepared by: Composable Security
Report ID: NRL-aaa33c8e
Test time period: 2025-09-15 - 2025-09-26
Retest time period: 2025-10-08 - 2025-10-09
Report date: 2025-10-10
Version: 1.1
Visit: composable-security.com

Contents

1. Retest summary (2025-10-09)



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 2025-09-15 and 2025-09-26) were correctly removed and no longer appear in the code.

The current status of the detected issues is as follows:

- the vulnerability with a **medium** impact on risk has been fixed,
- all vulnerabilities with a **low** impact on risk have been fixed,
- all security **recommendations** have been implemented.

The team has prepared the fixings in a very clear way, with a separate commit for each vulnerability and recommendation.

1.2. Scope

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

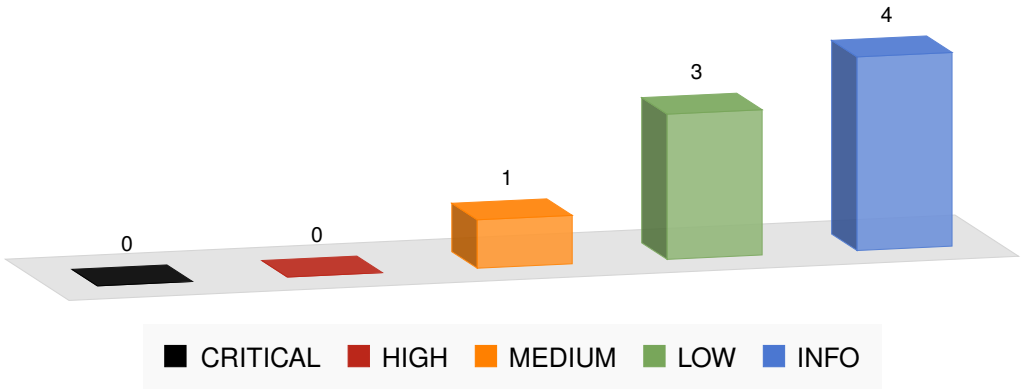
GitHub repository: https://github.com/Neverland-Money/neverland-contracts
CommitID: a4f40b7594c969330aa42d060f2c6461d5671c36

2. Current findings status

ID	Severity	Vulnerability	Status
NRL-aaa33c8e-M01	MEDIUM	Back-running reward notification	FIXED
NRL-aaa33c8e-L01	LOW	Vulnerable dependencies	FIXED
NRL-aaa33c8e-L02	LOW	Lack of 2-step ownership transfer	FIXED
NRL-aaa33c8e-L03	LOW	Reward not claimed by the reward receiver	FIXED

ID	Severity	Recommendation	Status
NRL-aaa33c8e-R01	INFO	Detect duplicated parameters	IMPLEMENTED
NRL-aaa33c8e-R02	INFO	Consider adding rewards receiver to splitted tokens	IMPLEMENTED
NRL-aaa33c8e-R03	INFO	Consider disabling renounceOwnership()	IMPLEMENTED
NRL-aaa33c8e-R04	INFO	Cache the array length outside a loop	IMPLEMENTED

3. Security review summary (2025-09-26)



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. This is the second iteration that focuses on new elements of self-repaying loans and rechecks the entire code as the last audit revealed a lot of vulnerabilities and many changes were introduced.

3.2. Results

The **Neverland Money** engaged Composable Security to review security of **Neverland Money**. Composable Security conducted this assessment over 2 person-week with 2 engineers.

The summary of findings is as follows:

- 1 vulnerability with a **medium** impact on risk was identified.
- 3 vulnerabilities with a **low** impact on risk were identified.
- 4 **recommendations** have been proposed that can improve overall security and help implement best practice.
- In addition to fixing vulnerabilities, the team is also listening to advice on how to improve security. Given that the first iteration uncovered numerous vulnerabilities, they decided to conduct a second round to ensure that their users are safe.
- Significant code improvements and best practices have been observed since the last audit.

- The team was engaged and the communication was very good.
- No vulnerabilities with critical and high impact on risk were identified.

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.

3.3. Centralization Risk

The current system implementation is not fully decentralized, allowing critical operations (e.g., changes of contracts' parameters) to be performed without additional security measures such as timelocks, granular permissions or multi-step processes. This is the result of the chosen architectural model and the intended behavior. It should not be treated as a vulnerability, but rather a risk that must be properly handled.

Sample functions that have a significant impact and can affect the project immediately:

- `setExecutor`
- `setSupportedAggregators`
- `setMaxSwapSlippageBps`
- `pause`
- `unpause`
- `emergencyWithdrawal`
- `setTransferStrategy`
- `configureAssets`
- `setEarlyWithdrawPenalty`
- `setEarlyWithdrawTreasury`
- `setMinLockAmount`
- `setRevenueReward`

This poses risks to the project's security, as it places a substantial amount of trust in the `Owner`, `EMISSION_MANAGER` and `team` roles. It is crucial to ensure the highest level of protection for the private keys associated with this roles.

Apart from the instant changes introduced by the function, some of them lack of parameters validation that could lead to either theft or Denial of Service. For example, the `setMaxSwapSlippageBps` function does not validate the new value of max slippage. It can become over 100% and allow tokens to be stolen.

While the project plans to share important operations via a multi-signature (multi-sig) setup, this does not fully address the underlying issue. Operations can still be introduced suddenly

without warning users, and the controlling parties retain complete control over user funds. Even though the code does not indicate any malicious intent, users should be aware of their dependency.

Recommendation

- To mitigate the risks associated with single points of failure or potential compromises, appropriate security requirements and procedures should be established to limit the impact in the event of loss of access or key compromise.
- Add parameters validation to administrative functions to set reasonable boundaries.
- Consider using `TimelockController` contract to delay changes of important parameters that can influence users's funds.
- Use multi-sig wallets to control the protocol contracts and require to initiate any critical updated by those wallets.

3.3.1 References

1. Secure Private Key Management for DApps
2. SCSVS G2: Policies and procedures
3. SCSVS G1: Architecture, design and threat modeling

3.4. Scope

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

GitHub repository: https://github.com/Neverland-Money/neverland-contracts
CommitID: aaa33c8e6f2b046e5954b50ac00ec82ae23c1151

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: aaa33c8e6f2b046e5954b50ac00ec82ae23c1151

Files in scope:

```
└── src
    ├── emissions
    │   ├── DustLockTransferStrategy.sol
    │   ├── DustRewardsController.sol
    │   └── DustTransferStrategyBase.sol
    ├── libraries
    │   ├── BalanceLogicLibrary.sol
    │   ├── CommonChecksLibrary.sol
    │   ├── CommonLibrary.sol
    │   ├── EpochTimeLibrary.sol
    │   └── SafeCastLibrary.sol
    ├── rewards
    │   └── RevenueReward.sol
    ├── self-repaying-loans
    │   ├── UserVault.sol
    │   ├── UserVaultFactory.sol
    │   └── UserVaultRegistry.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-aaa33c8e-M01] Back-running reward notification

MEDIUM FIXED

Retest (2025-10-08)

The vulnerability has been fixed by forbidding the increase of token amount for locks that are set to expire before the minimum duration.
The other mentioned cases are by design. As the Neverlan team states: "the protocol welcomes more locks based on upcoming revenue distribution, whether on the first or last day of ongoing the epoch."

Affected files

- RevenueReward.sol#L143

Description

The `DuctLock` contract currently permits a new reward to be added to the next epoch when the `notifyRewardAmount` function is invoked. This allows individuals to create locks before the new epoch and subsequently claim a portion of the newly added reward. The potential for exploitation increases with larger reward amounts due to the ability to optimize the timing and size of token locks.

Attackers can optimise the attack by initially creating small locks at the beginning of each epoch and subsequently increasing the amount of tokens in locks that are closest to expiration after a significant reward is added.

Attack scenario

Attackers may execute the following steps:

① Initiate locks with a minimal amount weekly with a month duration (greater durations lead to larger reward theft).
② Upon a substantial reward addition, increase the token amount in the lock nearing expiration just before the current epoch concludes.
③ Claim rewards in the following epoch.
④ Due to the increased token amount in the lock, a significant portion of the added reward is obtained.

Note: Attackers may also create a new lock after step 2, but a 4-week waiting period is

required to withdraw tokens.

Result: Part of the rewards are stolen.

Recommendation

Implement the following changes:

- Restrict the increase of token amounts for locks that are set to expire before the minimum duration.
- Modify the lock creation process to allow claiming rewards from the subsequent epoch instead of the current one.
- Assess the possibility of smoothing out the rewards deposited into the `RevenueReward` contract.

References

1. SCSVS G4: Business Logic

[NRL-aaa33c8e-L01] Vulnerable dependencies

LOW FIXED

Retest (2025-10-08)

The vulnerability has been fixed. The dependencies have been updated.

Affected files

- package.json

Description

Based on the provided NPM audit report, the project has a total of 20 vulnerabilities: 18 low severity, one high severity, and one critical severity. These are the most urgent issues requiring your attention.

- form-data (Critical)
- - Vulnerability: The form-data library, which is a transitive dependency (meaning it's not a direct dependency but is used by one of your direct dependencies), has a critical vulnerability. It uses an unsafe random function to generate the boundary for multipart requests.
- - Status: A fix is available (fixAvailable: true).
- axios (High)

- - Vulnerability: The axios library has a high severity vulnerability. The audit found two issues:
- - Server-Side Request Forgery (SSRF) and Credential Leakage: This is tracked as GHSA-jr5f-v2jv-69x6 and affects versions before 1.8.2. It could allow an attacker to make axios send requests to internal network resources or leak credentials.
- - Denial of Service (DoS): This is tracked as GHSA-4hjh-wcwx-xvwj and affects versions before 1.12.0. The lack of a data size check could be exploited to cause a DoS attack.
- - Status: A fix is available (fixAvailable: true).

Attack scenario

The attackers might take the following steps in turn:

① Analyze the dependencies used.
② Detect a known vulnerability.
③ Use a ready-made exploit.

Result of the attack: Depending on the vulnerability and its exploitability. A thorough analysis of the specific vulnerabilities has not been conducted due to the limited time and ease of remediation.

Recommendation

1. **Prioritize Fixes:** Address the critical and high severity vulnerabilities. Run npm audit fix to automatically update packages with available fixes. For axios and form-data, since fixAvailable is true, this command should resolve the issues.
2. **Manual Updates:** For packages where fixAvailable is false (e.g., several @nomicfoundation packages and cookie), you may need to manually update them to a version that addresses the vulnerability, or check if an upgrade path is available from the package maintainers.
3. **Dependency Review:** The report indicates that many vulnerabilities are in development dependencies (dev dependencies: 758 total), which are not used in production. While this lowers the risk for a deployed application, it's still good practice to fix them to ensure a secure development environment.
4. **Integrate a CI/CD Pipeline:** Incorporate npm audit and npm audit fix into your Continuous Integration/Continuous Deployment (CI/CD) pipeline. Run an audit on every commit or pull request, preventing new vulnerabilities from being merged into your main branch. Utilize dependency management tools like Dependabot (for GitHub).

References

1. SCSVS G1: Architecture, design and threat modeling

2. Dependabot quickstart guide

[NRL-aaa33c8e-L02] Lack of 2-step ownership transfer

LOW FIXED

Retest (2025-10-08)

The vulnerability has been fixed as recommended.

Affected files

- UserVaultRegistry.sol#L15

Description

Some smart contracts currently implement inheritance from `Ownable`. This pattern poses a risk of irretrievable loss of administrative control in case of instant transfer of ownership to any address except `address(0)`.

In case of invalid address passed as the new owner (e.g. a contract that cannot make calls to protected functions and the `transferOwnership` function) there is no way to get the ownership back.

To enhance security, it is recommended to transition to `Ownable2Step`, which offers a more secure mechanism for the transfer of elevated privileges through a structured two-step process.

Vulnerable scenario

The following steps lead to the described result:

① The owner transfers ownership to a new address that has a typo.
② The ownership is transferred instantly.
③ The owner cannot control the contract anymore, neither with old address because the ownerships has been transferred, nor with new one because they don't have access to the private key.

Result: Loss of the ownership for the contract.

Recommendation

- Inherit from `Ownable2Step` instead of `Ownable`.

References

1. Ownable2Step
2. SCSVS G1: Architecture Design Threat Modeling
3. SCSVS G5: Access Control

[NRL-aaa33c8e-L03] Reward not claimed by the reward receiver

LOW FIXED

Retest (2025-10-08)

The vulnerability has been fixed for selected notifications, excluding token burn and transfer. In those cases, the reward is claimed to the former owner of the token.

Affected files

- RevenueReward.sol#L184
- RevenueReward.sol#L196
- RevenueReward.sol#L207
- RevenueReward.sol#L228

Description

The revenue reward system permits claims by the token's owner, an authorized address, or via a vault created with the `enableSelfRepayLoan` function. Ideally, when a vault is designated, rewards should consistently be allocated to that vault. However, in the current implementation, the notification functions erroneously allocate rewards to the token owner instead of the intended rewards receiver.

Vulnerable scenario

The following steps lead to the described result:

① The owner of token 1 activates self-repay functionality.
② Rewards for token 1 are claimed and directed to the owner's vault.
③ The token undergoes a burning, transfer, merge, or split operation.
④ During the operation from point 3, the rewards are claimed for token 1, but they are incorrectly sent to the owner instead of the owner's vault.

Result: Rewards are improperly sent to the owner's address rather than to the owner's vault.

Recommendation

Modify the notification functions to correctly identify rewards receivers and transfer rewards accordingly. This can be achieved through the `_resolveRewardsReceiver` function since the `_removeToken` function, which clears the `tokenRewardReceiver` mapping, has not been executed at that point.

References

1. SCSVS G4: Business Logic

6. Recommendations

[NRL-aaa33c8e-R01] Detect duplicated parameters

INFO IMPLEMENTED

Retest (2025-10-08)

The recommendation has been implemented as recommended.

Description

The reward view functions (RevenueReward.sol#L471-L478, RevenueReward.sol#L481-L504, RevenueReward.sol#L507-L512) responsible for calculating rewards for DustLock tokens do not validate unique entries for tokenIds or reward tokens. This oversight allows for the potential duplication of rewards associated with the same tokens, resulting in inflated reward amounts.

This issue does not affect the claiming process, as `lastEarnTime` is updated for each token. However, it may lead to inaccuracies in financial reporting for projects that integrate with Neverland.

Recommendation

Implement a check to prevent the inclusion of duplicate DustLock tokenIds or reward tokens. A straightforward method is to require the list to be sorted in ascending order and to revert if the subsequent tokenId or token address is less than or equal to the previous one.

References

1. SCSVS G4: Business Logic

[NRL-aaa33c8e-R02] Consider adding rewards receiver to splitted tokens

INFO IMPLEMENTED

Retest (2025-10-08)

The recommendation has been implemented as recommended.

Description

During the token splitting process, the new tokens retain the same parameters, including mint time. However, the rewards receiver for the original token is cleared and not assigned to the newly created tokens:

- RevenueReward.sol#L220-L249

Recommendation

Ensure that after a token split, the rewards receiver is assigned from the original token to the newly minted tokens.

References

1. SCSVS G1: Architecture, design and threat modeling

[NRL-aaa33c8e-R03] Consider disabling renounceOwnership()

INFO IMPLEMENTED

Retest (2025-10-08)

The recommendation has been implemented as recommended.

Description

If the project does not plan to fully resign from admin control, leaving `renounceOwnership()` enabled can brick governance accidentally or via social engineering.

- UserVaultRegistry.sol#L15
- Dust.sol

Recommendation

Override `renounceOwnership()` to revert.

References

1. SCSVS G1: Architecture, design and threat modeling

[NRL-aaa33c8e-R04] Cache the array length outside a loop

INFO IMPLEMENTED

Retest (2025-10-08)

The recommendation has been implemented as recommended.

Description

Caching the array length outside a loop saves reading it on each iteration, as long as the array's length is not changed during the loop.

- UserVault.sol#L92
- DustRewardsController.sol#L112-L113
- DustRewardsController.sol#L249-L250
- DustRewardsController.sol#L287
- DustRewardsController.sol#L337

Recommendation

Cache the length outside the loop (e.g. `uint256 len = tokenIds.length;`).

References

1. SCSVS G10: Gas usage & limitations

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