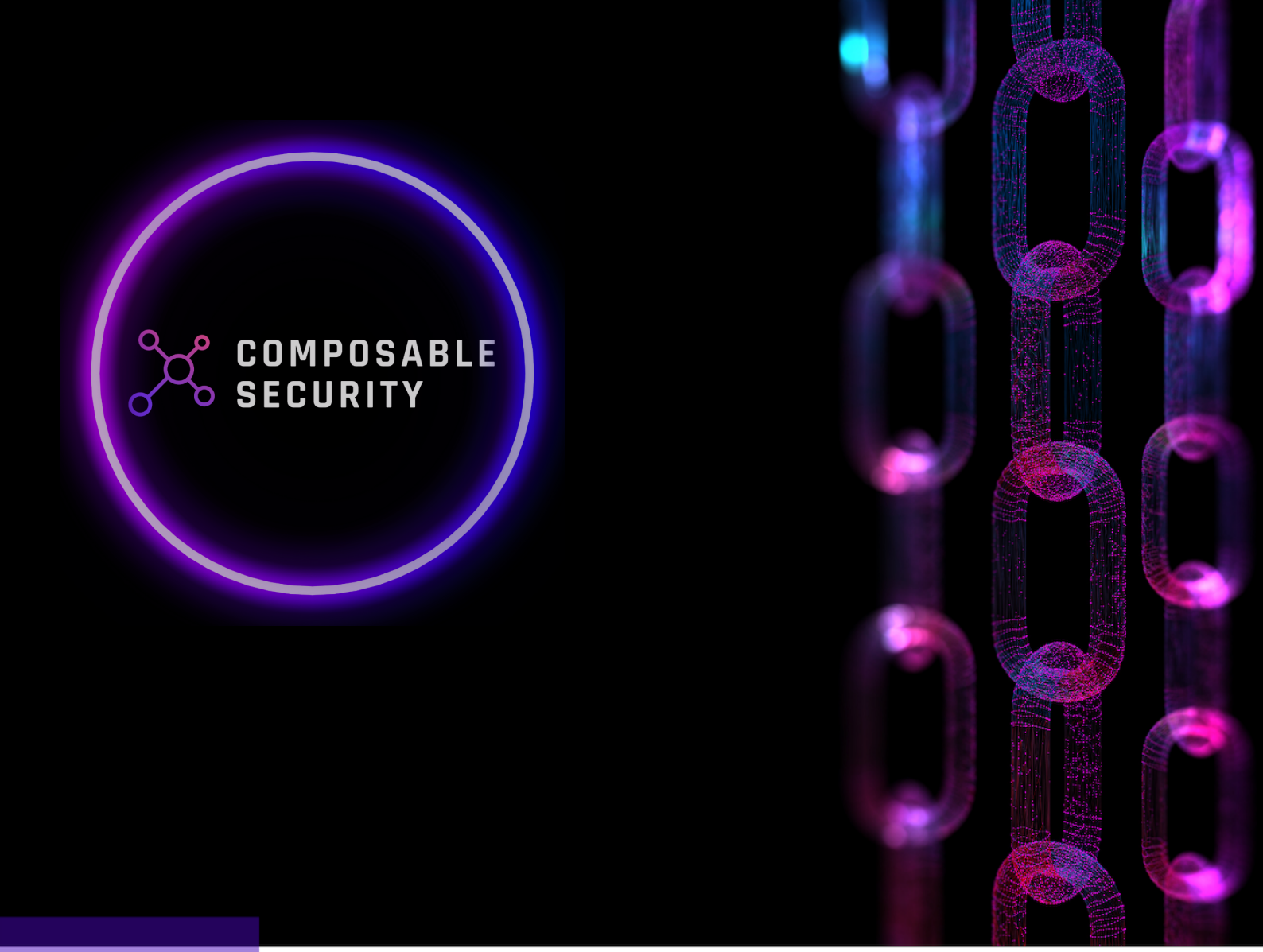


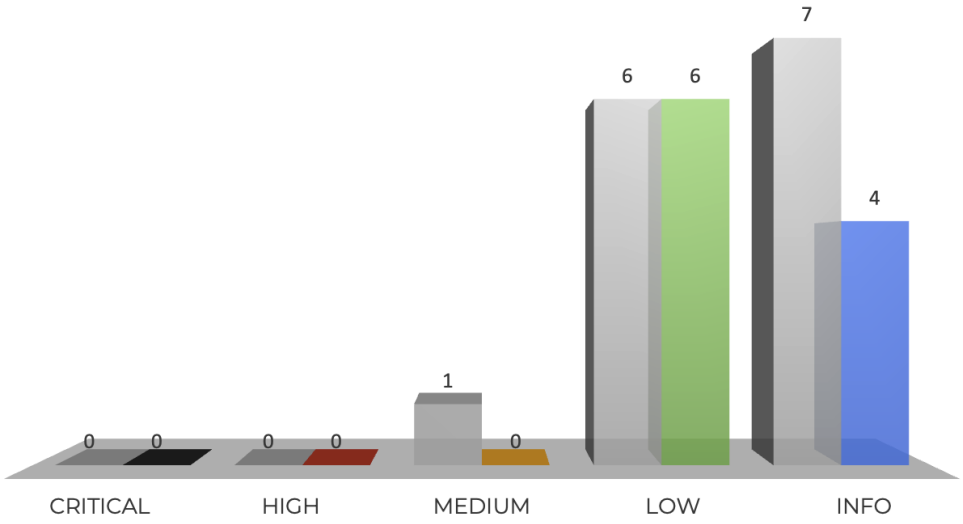
REPORT

Smart contract security review for Shroomy

Prepared by: Composable Security
Report ID: SHRM-f9d2b88d
Test time period: 2025-05-07 - 2025-05-14
Retest time period: 2025-05-19 - 2025-05-20
Report date: 2025-05-20
Version: 1.1
Visit: composable-security.com

Contents

1. Retest summary (2025-05-20)



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 2025-05-07and 2025-05-14) were correctly removed and no longer appear in the code.

The current status of the detected issues is as follows:

- 1 vulnerability with a **medium** impact on risk has been fully removed.
- 6 vulnerabilities with a **low** impact on risk have been acknowledged by the team.
- 7 security **recommendations** were handled as follows:
  - 3 have been implemented,
  - 4 have been acknowledged.

The team prioritized resolving the most important bug in the code but chose not to address smaller issues that, while less severe, still influence best security practices. This decision was primarily driven by considerations of gas efficiency and the project's centralized architecture, which enables the delegation of certain responsibilities to other components.

For future development cycles, it is strongly recommended to implement all reported recommendations to enhance the protocol's resilience, reduce systemic risk, and move toward a more robust and anti-fragile design.

1.2. Scope

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

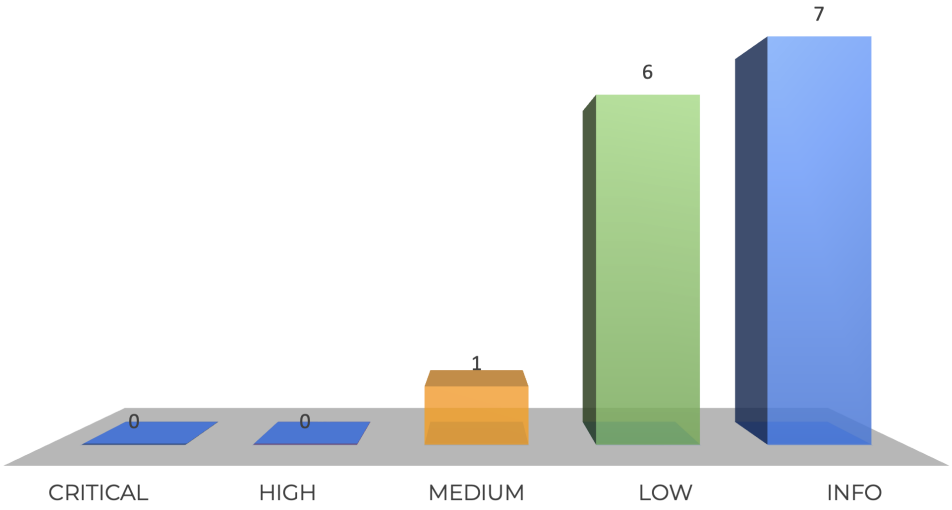
GitHub repository: https://github.com/23stud-io/aave-deploy-v3/
CommitID: f9d2b88d85b9c155cd0ff878a516ca8d002e3499 (not changed)

GitHub repository: https://github.com/23stud-io/contracts.shroomy.staking/
CommitID: 48f5000501f6b653eb91fe68c5b19720188efb00

2. Current findings status

ID	Severity	Vulnerability	Status
SHRM-f9d2b88d-M01	MEDIUM	Inconsistent state via reentrancy	FIXED
SHRM-f9d2b88d-L01	LOW	Lack of two-step ownership transfer	ACKNOWLEDGED
SHRM-f9d2b88d-L02	LOW	Rewards and tokens loss on reverting reward token	ACKNOWLEDGED
SHRM-f9d2b88d-L03	LOW	Denial of Service due to Out of Gas error	ACKNOWLEDGED
SHRM-f9d2b88d-L04	LOW	Invalid revenue amount transferred	ACKNOWLEDGED
SHRM-f9d2b88d-L05	LOW	Arbitrary call to swapper	ACKNOWLEDGED
SHRM-f9d2b88d-L06	LOW	Revert on approvals	ACKNOWLEDGED
ID	**Severity**	**Recommendation**	**Status**
SHRM-f9d2b88d-R01	INFO	Use custom errors	ACKNOWLEDGED
SHRM-f9d2b88d-R02	INFO	Emit events for important state changes	ACKNOWLEDGED
SHRM-f9d2b88d-R03	INFO	Remove unnecessary code	IMPLEMENTED
SHRM-f9d2b88d-R04	INFO	Implement full metadata interface	ACKNOWLEDGED
SHRM-f9d2b88d-R05	INFO	Change comparison sign	IMPLEMENTED
SHRM-f9d2b88d-R06	INFO	Remove unused ERC721EnumerableForbiddenBatchMint custom error	IMPLEMENTED
SHRM-f9d2b88d-R07	INFO	Consider using newest Solidity version	ACKNOWLEDGED

3. Security review summary (2025-05-14)



3.1. Client project

The Shroomy project is a fork of AAVE v3 on Ink with additional contracts that support staking tokens with various lock periods and earning rewards.

The project consists of two repositories. One contains the revenue distribution implementation, and the other contains the staking implementation.

3.2. Results

Composable Security was engaged to review the security of Shroomy. Composable Security conducted this assessment over 1 person-week with 2 engineers.

The summary of findings is as follows:

- **One** vulnerability with a **medium** impact on risk was identified.
- **Six** vulnerabilities with a **low** impact on risk were identified.
- Seven **recommendations** have been proposed that can improve overall security and help implement best practice.
- The project is highly centralized with the Owner role than is very powerful. The details have been presented in section 3.3.
- The team was engaged and the communication was very good.

Composable Security recommends that **Shroomy** 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.

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

- `setTreasury`
- `setRevenueProcessor`
- `setTreasuryPercentage`
- `setPoolAddress`
- `setVault`
- `setRewardToken`
- `approveToken`
- `rescueToken`
- `setOperatorStatus`

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

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, the 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.
- 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/23stud-io/aave-deploy-v3/
CommitID: f9d2b88d85b9c155cd0ff878a516ca8d002e3499

GitHub repository: https://github.com/23stud-io/contracts.shroomy.staking/
CommitID: 12986193c396bf94dc74622abd47474eafdf9663

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 **Shroomy** 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 **Shroomy** repository.

GitHub repository:
https://github.com/23stud-io/aave-deploy-v3/
CommitID: f9d2b88d85b9c155cd0ff878a516ca8d002e3499

Files in scope:

```
.
└── contracts
    ├── abstract
    │   └── OperatorControlled.sol
    ├── RevenueDistributor.sol
    ├── RevenueProcessor.sol
    └── Vault.sol
```

GitHub repository: https://github.com/23stud-io/contracts.shroomy.staking/
CommitID: 12986193c396bf94dc74622abd47474eafdf9663

Files in scope:

```
.
└── contracts
    ├── LockerPass.sol
    └── ShroomyLocker.sol
```

Documentation: The behavior of the contracts was described in the comments, no additional documentation was provided.

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:

- aTokens.
- Reward Tokens.
- Staking Positions.

Potential attackers goals:

- Theft of tokens.
- Lock users' tokens in the contract.
- Treasury percentage manipulation.
- Denial of Service

Potential scenarios to achieve the indicated attacker's goals:

- Unauthorized claim for specific staking position.
- Unauthorized minting of tokens.
- Unauthorized unstaking.
- 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.
- Unintentional loss of the ability to govern the system.
- Private key compromise, rug-pull.

4.4. Testing methodology

Smart contract security review was performed using the following methods:

- Q&A sessions with the **Shroomy** 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

[SHRM-f9d2b88d-M01] Inconsistent state via reentrancy

MEDIUM FIXED

Retest (2025-05-19)

The team implemented all recommendations and additionally created a test analyzing this scenario. The clearance of token approvals is done in LockerPass.sol#L197. The `onERC721Received` is now invoked after completing the transfer logic: LockerPass.sol#L294, LockerPass.sol#L311.

Affected files

- LockerPass.sol#L292-L295
- ShroomyLocker.sol#L153-L168

Description

The implementation of the `safeTransferFrom` function executes the external call to the recipient before completing the transfer logic. This sequence creates a vulnerability due to inadequate approval and ownership checks, allowing an attacker to unstake while retaining ownership of a LockerPass.

Note: It should be emphasized that the staking position associated with the LockerPass is cleared during this process, and an attacker cannot steal tokens by attempting to unstake from the same position again.

Attack scenario

Attackers can exploit this vulnerability through the following steps:

① Stake tokens to obtain a LockerPass token.
② Transfer the LockerPass token to their own contract.
③ The contract sends an approval to itself for the LockerPass token and subsequently calls the `safeTransferFrom` function, transferring the token from the zero address to itself.
④ The LockerPass contract triggers the `onERC721Received` function within the attacker's contract, which then calls the `unstake` function.
⑤ The `unstake` function burns the LockerPass token, and at this point, the transfer logic begins execution.

⑥ The `transferFrom` function (the transfer logic) verifies that the current owner is the zero address and confirms that the approved operator is the attacker's contract (which has not been cleared upon burning), thereby changing the owner back to the attacker's contract.
⑦ The attacker's contract successfully unstakes the tokens while still maintaining ownership of the LockerPass token.

A proof of concept (PoC) has been shared with the development team.

Result of the attack: Inconsistent state where the burnt tokens appear to be owned by users.

Recommendation

To address this vulnerability, the following steps should be implemented:

- Clear `_tokenApprovals` for the token upon burning.
- Invoke the `onERC721Received` function after completing the transfer logic.

References

1. SCSVS G6: Communication

[SHRM-f9d2b88d-L01] Lack of two-step ownership transfer

LOW ACKNOWLEDGED

Retest (2025-05-19)

The team is aware and has decided not to implement a fix for this issue for the following reasons: "We are not implementing the fix for two-step ownership transfer as the contract will be managed by a multisig. Ownership transfer will be verified before the project handover".

Affected files

- ShroomyLocker.sol#L18
- Vault.sol#L8
- OperatorControlled.sol#L11

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`.
- Consider overriding the `renounce()` function if there is no intention to forfeit control completely.

References

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

[SHRM-f9d2b88d-L02] Rewards and tokens loss on reverting reward token

LOW ACKNOWLEDGED

Retest (2025-05-19)

The team is aware and has decided not to implement a fix for this issue for the following reasons: "The project team will carefully select reward tokens, using only verified and secure contracts".

Affected files

- ShroomyLocker.sol#L232

Description

The `_claim` function is responsible for iterating through all reward tokens to distribute rewards to the owner of a stake position. If an error occurs with the transfer of any single reward token, the entire claiming process fails, preventing the user from receiving any rewards.

Furthermore, the `unstake` function is designed to automatically call the `_claim` function to ensure all rewards are collected before allowing an unstake. As a result, if any reward token encounters an issue during the transfer, users are unable to unstake their tokens, effectively locking both staked tokens and rewards in the contract.

There is no functionality that permits the removal of a problematic reward token, meaning that the only solution to this issue lies in modifying the implementation of the reward token itself.

Vulnerable scenario

This issue can occur through the following sequence:

① A user stakes tokens in the contract.
② Over time, the user accumulates rewards from multiple reward tokens.
③ When the user attempts to unstake their tokens, the `_claim` function fails due to an issue with the transfer of one of the reward tokens.
④ As a result, both the staked tokens and accrued rewards remain locked in the contract.

Result: Users' rewards and staked tokens become inaccessible and remain locked in the staking contract.

Recommendation

To enhance user experience and mitigate this issue, consider decoupling the reward claiming process from the unstaking process. This would allow users to unstake their tokens independently of claiming rewards.
Furthermore, implementing a feature that lets users claim specific reward tokens would provide additional flexibility and reduce the risk of being locked out due to a single failing token.

References

1. SCSVS G8: Denial of service

[SHRM-f9d2b88d-L03] Denial of Service due to Out of Gas error

LOW ACKNOWLEDGED

Retest (2025-05-19)

The team is aware and has decided not to implement a fix for this issue for the following reasons: "In case of a large number of assets, authorized operators can call `distributeToken` for specific tokens individually. Support for new assets is a controlled process and we do not expect a sudden, significant increase in their number".

Affected files

- RevenueDistributor.sol#L132
- RevenueProcessor.sol#L78
- RevenueProcessor.sol#L101
- RevenueProcessor.sol#L183

Description

Some loops iterate over the unbound lists, retrieved from other contracts (e.g. the list `supportedAssets` retrieved from the pool using the `getReservesList` function). If the list returned by those function grows significantly, it can lead to reverts due to the Out of Gas errors, blocking the unwraping and distributions of tokens.

Vulnerable scenario

The following steps lead to the described result:

① The RevenueDistributor is associated to a pool that supports some tokens.
② The list returned by `getReservesList` is small enough that the `distribute` function is executed correctly and successfully.
③ The list of reserves increases significantly.
④ The next call to `distribute` function iterates over all reserves from `getReservesList` result and uses all gas.

Result: Revenue loss because of the locked tokens.

Recommendation

Consider allowing to iterate over selected indexes of the list.

References

1. SCSVS G8: Denial of service

[SHRM-f9d2b88d-L04] Invalid revenue amount transferred

LOW ACKNOWLEDGED

Retest (2025-05-19)

The team is aware and has decided not to implement a fix for this issue for the following reasons: "The distribute function is regularly called, ensuring that any remaining small amount of tokens will be distributed in the next cycle".

Affected files

- RevenueDistributor.sol#L108

Description

The `distributeToken` function retrieves the balance of token to be distributed using `scaledBalanceOf` function which returns smaller amount than the real balance of the distributor.

Note: The development team has addressed this in the comments and noted that "*It returns the internal scaled balance without interest, which is safer for arithmetic operations*" which is not true.

Result: Inability to distribute all tokens in one call to `distributeToken`.

Recommendation

Use the `balanceOf` function to retrieve the amount to be transferred.

References

1. SCSVS I2: Token

[SHRM-f9d2b88d-L05] Arbitrary call to swapper

LOW ACKNOWLEDGED

Retest (2025-05-19)

The team is aware and has decided not to implement a fix for this issue for the following reasons: "SuperSwap API provides encoded calldata, which significantly complicates their decoding and validation of parameters such as `minAmountOut`. The output token is strictly defined as `rewardToken` during calldata generation by authorized operators who verify parameters and monitor swap execution. Additionally, the slippage parameter is configured during data preparation in the Superswap API before calldata generation. If `minAmountOut` is not achieved, the transaction will be rejected".

Affected files

- RevenueProcessor.sol#L133

Description

The `batchExecuteRawSuperSwap` function uses low-level call to make a swap. However, it is an error-prone design where unspecified or zeroed minimum output amount can lead to the swap being sandwiched and the RevenueProcessor contract being drained.

Additionally, there is no check whether the last swap has the reward token as the output token.

Attack scenario

The following steps lead to the described result:

① The authorized address calls the `batchExecuteRawSuperSwap` function with `superSwapCalldatas` calldatas.
② One of the calldata does not specify the minimum output token amount.
③ The attacker notices sandwichable swap and imbalances the pool before swap.
④ The swap is being executed.
⑤ The attacker uses the second transaction to rebalance the pool and steal tokens swapped by the authorized address.

Result of the attack: Partial theft of swapped tokens using sandwiching.

Recommendation

- Use specific functions to execute swaps (e.g. via adapter contracts that integrate with particular protocol).
- Check whether the last output token is the reward token.
- Validate the final output token amount to be greater or equal to the minimum amount specified in the parameter.

References

1. SCSVS G4: Business Logic

[SHRM-f9d2b88d-L06] Revert on approvals

LOW ACKNOWLEDGED

Retest (2025-05-19)

The team is aware and has decided not to implement a fix for this issue for the following reasons: "The issue concerns non-standard ERC20 implementations. In case of encountering a problematic token, operators can use a specific approach to swap or introduce dedicated functions to exchange tokens for a dedicated reward token".

Affected files

- RevenueProcessor.sol#L130-L137

Description

Certain tokens do not permit setting approvals to zero when they are already at zero. This restriction can lead to a revert during the approval reset process at line 137. Similarly, these tokens also disallow setting approvals to a non-zero value if the current approval is already non-zero, which can result in a revert at line 130.

Vulnerable scenario

The following steps illustrate how the issue can occur:

① The authorized address aims to swap 100 tokens (utilizing an input token that reverts when attempts are made to change approvals from 0 to 0). It initiates a call to `batchExecuteRawSuperSwap` with the intended amount set as 100 (provided in the `superSwapCalldatas` parameter).
② The contract attempts to approve the transfer of 100 tokens and proceeds with the swap.
③ During the process, the contract tries to clear the approval, but since the current approval is already 0, the call to the `approve` function fails.
④ This failure cancels the swap.

Result: Inability to successfully swap the token.

Recommendation

Ensure that the approval is set for the amount being swapped and only attempt to clear the approval if there is an existing approval present.

References

1. SCSVS I2: Token

6. Recommendations

[SHRM-f9d2b88d-R01] Use custom errors

INFO ACKNOWLEDGED

Retest (2025-05-19)

For now, the team decided not to implement the recommended best practice.

Description

Custom errors offer enhanced gas efficiency compared to traditional string messages and enable developers to provide detailed descriptions of errors utilizing NatSpec documentation.

Furthermore, starting from version 0.8.26 of Solidity, it is no longer necessary to negate the `require` expression in order to revert when utilizing custom errors, as they are now directly compatible with the `require` function.

Currently, all `require` statements use string messages instead of custom ones.

Recommendation

Implement custom errors in place of string messages.
Custom errors have been supported by the `require` function since Solidity version 0.8.26.

```
require(
    balance[msg.sender] >= amount,
    InsufficientBalance(balance[msg.sender], amount)
);
```

References

1. SCSVS G11: Code Clarity

[SHRM-f9d2b88d-R02] Emit events for important state changes

INFO ACKNOWLEDGED

Retest (2025-05-19)

For now, the team decided not to implement the recommended best practice.

Description

The update of critical parameters should be tracked with events.

Recommendation

Emit an event in the following functions:

- `setPoolAddress`

References

1. SCSVS G1: Architecture, design and threat modeling
2. Principles and Best Practices to Design Solidity Events in Ethereum and EVM

[SHRM-f9d2b88d-R03] Remove unnecessary code

INFO IMPLEMENTED

Retest (2025-05-19)

The recommendation has been implemented as recommended.

Description

There are code snippets that have no effect on the business logic and cannot be reached:

```
124 require(msg.sender != address(0), 'Invalid address');
```

Recommendation

Remove indicated code snippets.

References

1. G11: Code clarity

[SHRM-f9d2b88d-R04] Implement full metadata interface

INFO ACKNOWLEDGED

Retest (2025-05-19)

For now, the team decided not to implement the recommended best practice.

Description

The LockerPass contract implements `symbol` and `name` function from ERC721Metadata interface, but it does not support this interface (in `supportsInterface` function) and does not implement `tokenURI` function.

Recommendation

Consider full implementation and support of ERC721Metadata interface.

References

1. SCSVS G1: Architecture, design and threat modeling
2. G11: Code clarity

[SHRM-f9d2b88d-R05] Change comparison sign

INFO IMPLEMENTED

Retest (2025-05-19)

The recommendation has been implemented as recommended.

Description

The `unstake` function checks whether the `unlockAt` is reached. However, it uses strict greater than sign which means that when the current timestamp is equal to `unlockAt` the tokens cannot be unstaked. The contract should allow to unstake tokens when the block timestamp is equal to `unlockAt`.

Recommendation

Change < to <=.

References

1. G11: Code clarity

[SHRM-f9d2b88d-R06] Remove unused ERC721EnumerableForbiddenBatchMint custom error

INFO IMPLEMENTED

Retest (2025-05-19)

The recommendation has been implemented as recommended.

Description

The LockerPass contract contains a custom error that is not used anywhere.

```
51 error ERC721EnumerableForbiddenBatchMint();
```

Recommendation

It is recommended to remove unused custom error.

References

1. G11: Code clarity

[SHRM-f9d2b88d-R07] Consider using newest Solidity version

INFO ACKNOWLEDGED

Retest (2025-05-19)

For now, the team decided not to implement the recommended best practice.

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:

```
pragma solidity ^0.8.20;
```

Recommendation

Use the latest stable version of major Solidity release:

```
pragma solidity 0.8.29;
```

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

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