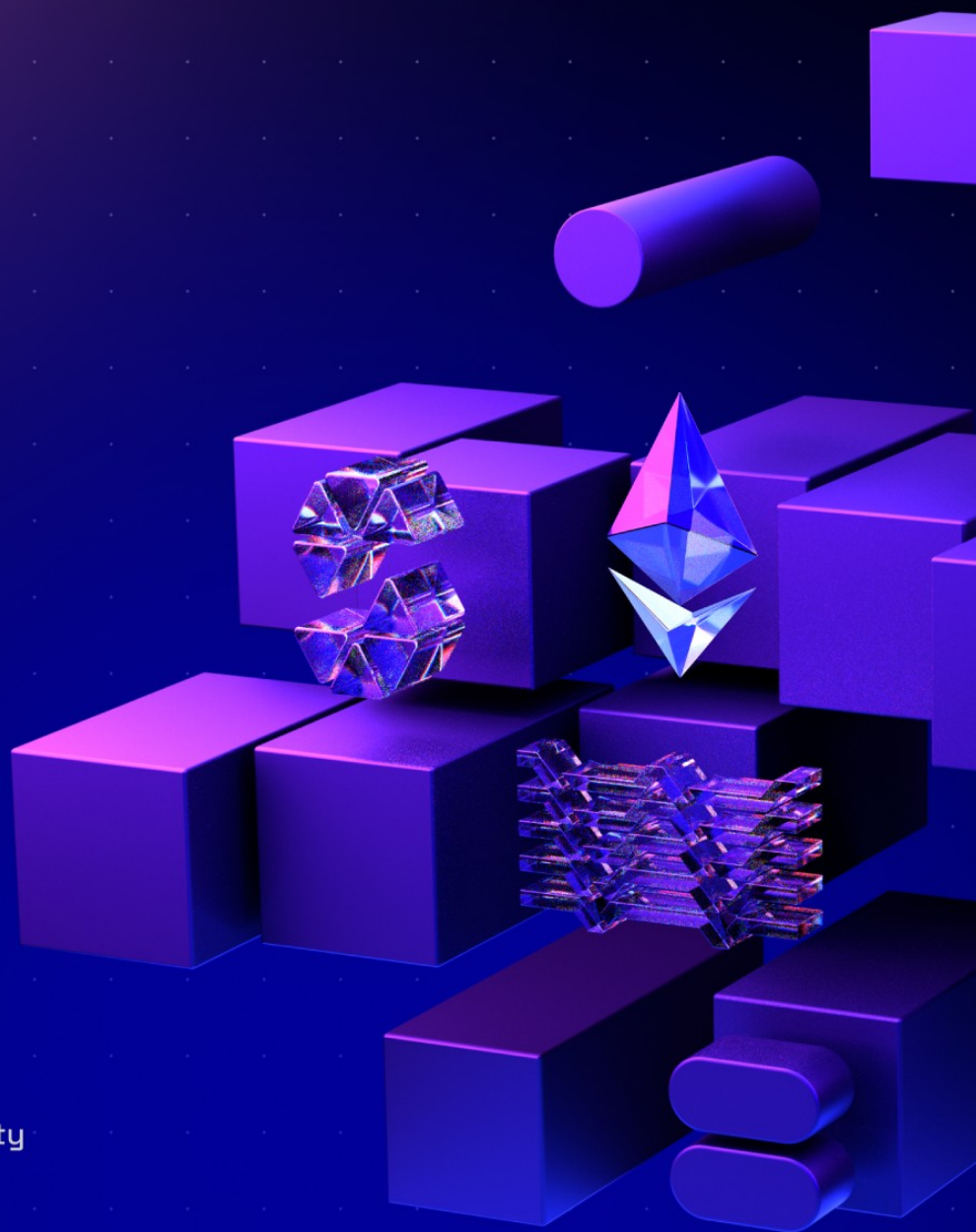
ackee
blockchain security

BreederDAO

Token migrator

22.11.2024



Ackee Blockchain Security

Contents

1. Document Revisions

1.0-draft	Draft Report	21.10.2024
1.1	Final Report	22.11.2024

2. Overview

This document presents our findings in reviewed contracts.

2.1. Ackee Blockchain Security

Ackee Blockchain Security is an in-house team of security researchers performing security audits focusing on manual code reviews with extensive fuzz testing for Ethereum and Solana. Ackee is trusted by top-tier organizations in web3, securing protocols including Lido, Safe, and Axelar.

We develop open-source security and developer tooling Wake for Ethereum and Trident for Solana, supported by grants from Coinbase and the Solana Foundation. Wake and Trident help auditors in the manual review process to discover hardly recognizable edge-case vulnerabilities.

Our team teaches about blockchain security at the Czech Technical University in Prague, led by our co-founder and CEO, Josef Gattermayer, Ph.D. As the official educational partners of the Solana Foundation, we run the School of Solana and the Solana Auditors Bootcamp.

Ackee's mission is to build a stronger blockchain community by sharing our knowledge.

Ackee Blockchain a.s.
Rohanske nabrezi 717/4
186 00 Prague, Czech Republic
https://ackee.xyz
hello@ackee.xyz

2.2. Audit Methodology

1. **Verification of technical specification**
   The audit scope is confirmed with the client, and auditors are onboarded to the project. Provided documentation is reviewed and compared to the audited system.

2. **Tool-based analysis**
   A deep check with Solidity static analysis tool Wake in companion with Solidity for VS Code extension is performed, flagging potential vulnerabilities for further analysis early in the process.

3. **Manual code review**
   Auditors manually check the code line by line, identifying vulnerabilities and code quality issues. The main focus is on recognizing potential edge cases and project-specific risks.

4. **Local deployment and hacking**
   Contracts are deployed in a local Wake environment, where targeted attempts to exploit vulnerabilities are made. The contracts' resilience against various attack vectors is evaluated.

5. **Unit and fuzz testing**
   Unit tests are run to verify expected system behavior. Additional unit or fuzz tests may be written using Wake framework if any coverage gaps are identified. The goal is to verify the system's stability under real-world conditions and ensure robustness against both expected and unexpected inputs.

2.3. Finding Classification

A *Severity* rating of each finding is determined as a synthesis of two sub-ratings: *Impact* and *Likelihood*. It ranges from *Informational* to *Critical*.

If we have found a scenario in which an issue is exploitable, it will be assigned an impact rating of *High*, *Medium*, or *Low*, based on the direness of the consequences it has on the system. If we haven't found a way, or the issue is only exploitable given a change in *configuration* (system settings or parameters, such as deployment scripts, compiler configurations, using multi-signature wallets for owners, etc.) or given a change in the codebase, then it will be assigned an impact rating of *Warning* or *Info*.

Low to *High* impact issues also have a *Likelihood*, which measures the probability of exploitability during runtime.

The full definitions are as follows:

Severity

		Likelihood			
		High	**Medium**	**Low**	**N/A**
Impact	**High**	Critical	High	Medium	-
	Medium	High	Medium	Low	-
	Low	Medium	Low	Low	-
	Warning	-	-	-	Warning
	Info	-	-	-	Info

Table 1. Severity of findings

Impact

- **High** - Code that activates the issue will lead to undefined or catastrophic consequences for the system.
- **Medium** - Code that activates the issue will result in consequences of serious substance.
- **Low** - Code that activates the issue will have outcomes on the system that are either recoverable or don't jeopardize its regular functioning.
- **Warning** - The issue cannot be exploited given the current code and/or *configuration*, but could be a security vulnerability if these were to change slightly. If we haven't found a way to exploit the issue given the time constraints, it might be marked as a "Warning" or higher, based on our best estimate of whether it is currently exploitable.
- **Info** - The issue is on the borderline between code quality and security. Examples include insufficient logging for critical operations. Another example is that the issue would be security-related if code or *configuration* was to change.

Likelihood

- **High** - The issue is exploitable by virtually anyone under virtually any circumstance.
- **Medium** - Exploiting the issue currently requires non-trivial preconditions.
- **Low** - Exploiting the issue requires strict preconditions.

2.4. Review Team

The following table lists all contributors to this report. For authors of the specific revision, see the “Revision team” section in the respective “Report revision” chapter.

Member’s Name	Position
Dmytro Khimchenko	Lead Auditor
Naoki Yoshida	Auditor
Josef Gattermayer, Ph.D.	Audit Supervisor

2.5. Disclaimer

We’ve put our best effort to find all vulnerabilities in the system, however our findings shouldn’t be considered as a complete list of all existing issues. The statements made in this document should not be interpreted as investment or legal advice, nor should its authors be held accountable for decisions made based on them.

3. Executive Summary

Revision 1.0

BreederDAO engaged Ackee Blockchain Security to perform a security review of the part of the BreederDAO protocol with a total time donation of 3 engineering days in a period between October 15 and October 21, 2024, with Dmytro Khimchenko as the lead auditor.

The audit was performed on the commit `51481a`[1] and the scope was the following:

- contracts/disperse-breed/DisperseBreed.sol
- contracts/sovrun-token/SovrunToken.sol
- contracts/token-migrator/token-migrator.sol

We began our review using static analysis tools, including Wake. The static analysis helped us to discover several issues, such as L1 and W3. We then took a deep dive into the logic of the contracts. We implemented manually-guided differential stateful fuzz tests in Wake testing framework to verify the correctness of computation during migration process, correctness of airdrop contract. More details on the fuzzing process can be found in Report revision 1.0 Fuzzing of the contracts yielded finding W2.

During the review we paid special attention to:

- ensuring the arithmetic of the system is correct,
- detecting possible reentrancies in the code,
- ensuring access controls are not too relaxed or too strict,
- ensuring tokens are compliant with EIP standards,
- airdrop functionality works as expected,

- looking for common issues such as data validation.

Our review resulted in 12 findings, ranging from Info to High severity. The most severe one is H1, because this finding in combination with L2 can lead to users taking leverage from arbitrage opportunities, that can result in arbitrageurs drain liquidity or deflation of the tokens.

Ackee Blockchain Security recommends BreederDAO:

- allow the conversion rate to support fractional values for greater precision,
- take advantage of the decimals of the token, which can help to avoid the precision issue,
- give the `TokenMigrator` contract the `BURNER_ROLE` role to burn the token and decrease `totalSupply` of token,
- address all other reported issues.

See Report Revision 1.0 for the system overview and trust model.

Revision 1.1

The review was done on the given commit `51f09b`[2]. The BreederDAO team fixed most of the issues and acknowledged others with comments, that can be seen in the report.

[1] full commit hash: `51481a446f74057231dba3c7c1af9c51ad67a7e3`

[2] full commit hash: `51f09b30707f5840f08fd3294f4fcde8d757fef0`

4. Findings Summary

The following section summarizes findings we identified during our review. Unless overridden for purposes of readability, each finding contains:

- *Description*
- *Exploit scenario* (if severity is low or higher)
- *Recommendation*
- *Fix* (if applicable).

Summary of findings:

Critical	High	Medium	Low	Warning	Info	Total
0	1	1	3	3	4	12

Table 2. Findings Count by Severity

Findings in detail:

Finding title	Severity	Reported	Status
H1: Precision misconfiguration	High	1.0	Fixed
M1: renounceOwnership locks token	Medium	1.0	Fixed
L1: Locked ether	Low	1.0	Fixed
L2: Fixed conversion rate arbitrage	Low	1.0	Acknowledged
L3: Two step ownership is not used	Low	1.0	Acknowledged
W1: Event emits hash	Warning	1.0	Fixed

Finding title	Severity	Reported	Status
W2: Burn is not used during token migration	Warning	1.0	Acknowledged
W3: Unused safeERC20	Warning	1.0	Fixed
I1: Check approval once to save gas	Info	1.0	Fixed
I2: Natspec is missing	Info	1.0	Fixed
I3: Inconsistent filename	Info	1.0	Fixed
I4: String error used instead of custom error	Info	1.0	Fixed

Table 3. Table of Findings

Report Revision 1.0

Revision Team

Member's Name	Position
Dmytro Khimchenko	Lead Auditor
Naoki Yoshida	Auditor
Josef Gattermayer, Ph.D.	Audit Supervisor

System Overview

The Breeder DAO project has smart contracts for airdrop distribution and token migration. Token migration contract is used to migrate Breeder Token to Sovrun Token with a fixed conversion rate to ensure a predictable swap mechanism. This contract includes a circuit breaker feature, which allows the Breeder DAO to halt migrations at any time and mitigate risks associated with market volatility. The reason for this is the potential difference between the price of Breeder Token and Sovrun Token on the market.

Trust Model

The smart contracts in the scope of the audit are permissionless and have small trust assumptions. The `DisperseBreed.sol` contract is designed to be primarily permissionless. The only bit of admin involvement is with the `withdraw` function, which lets the owner recover tokens accidentally sent to the contract. While this is a safeguard to prevent a loss of tokens, this functionality assumes that users trust the admins not to steal the tokens. The owner of the `SovrunToken.sol` contract can assign `MINTER_ROLE` and `BURNER_ROLE` to any address, meaning those addresses will have the power to mint new tokens or burn existing ones. This implies that the community must trust these roles will only be given to reliable addresses and will not be

misused. The owner of the `token-migrator.sol` contract has the authority to stop the migration process whenever they consider it necessary. This gives flexibility to protect the system from market fluctuations or potential risks.

Fuzzing

A manually-guided differential stateful fuzz test was developed during the review to test the correctness and robustness of the system.

The differential fuzz test keeps its own Python state according to the system's specification. Assertions are used to verify the Python state against the on-chain state in contracts.

The list of all implemented execution flows and invariants is available in Appendix B.

The fuzz tests simulate the whole system and make strict assertions about the behavior of the contracts.

The full source code of all fuzz tests is available and can be shared to the client.

Findings

The following section presents the list of findings discovered in this revision.

H1: Precision misconfiguration

High severity issue

Impact:	High	Likelihood:	Medium
Target:	token-migrator.sol	Type:	Logic error

Description

The `TokenMigrator` contract restricts the conversion rate (`_convRate`) to integer values by requiring it to be divisible by `1e18` without remainder. This configuration limits the precision of the conversion rate, which is suboptimal for ERC-20 tokens that typically **support up to 18 decimal places**. As a result, it cannot perform migrations with fractional conversion rates, such as converting a source token amount to 0.9 of a destination token.

Listing 1. Excerpt from token-migrator

```
63 if (_convRate == 0 || _convRate % 1e18 != 0) {
64     revert InvalidConversionRate();
65 }
```

Exploit scenario

1. `TokenMigrator` contract is deployed with logical conversion rate of 1.
2. After a period of time, SovrunToken price is 1.5 times less than BreederToken. It results to the situation, where users do not have interest in migrating their BreederTokens to SovrunTokens.
3. BreederDAO cannot create a new migration contract with logical conversion rate of 1.5 to create interest for users, because it is not possible to set it with integer value.

Recommendation

There are two different ways how the problem can be addressed:

- Implement the check in the `constructor` function, which is not as restrictive as the current implementation.

```
constructor(address _srcToken, address _destToken, uint256 _convRate) {
    // <--SNIP-->

    if (_convRate == 0 || _convRate % 1e16 != 0) {
        revert InvalidConversionRate();
    }

    // <--SNIP-->
}
```

This change will not lead to precision loss, as BreederToken and SovrunToken have 18 decimal precision, which is enough to be protected from precision loss.

- Use a lower value of `conversionRate` (e.g., 100 for 1:1 conversion rate, 150 for 1:1.5 conversion rate) instead of 1e18. Remove the `_convRate % 1e18 == 0` condition in the constructor and only check if `convRate` does not equal 0.

```
constructor(address _srcToken, address _destToken, uint256 _convRate) {
    // <--SNIP-->

    if (_convRate == 0) {
        revert InvalidConversionRate();
    }

    // <--SNIP-->
}
```

Besides, ensure that user provided `_srcTokenAmt % 1e2 == 0`. In other case, it will lead to precision loss.

```
function migrate(uint256 _srcTokenAmt) external isMigrationEnabled {
    // <--SNIP-->
    if (_srcTokenAmt % 1e2 != 0) revert InvalidUnitAmount();
    uint256 destTokenAmt = (_srcTokenAmt * convRate) / 1e2;
    // transfer the amount to 0xdead address / burn token
    // <--SNIP-->

}
```

Fix 1.1

The issue was fixed by modifying the `_convRate` precision handling to have 2 decimal places. The maximum precision loss in this configuration is limited to `100 / 10**18`, ensuring that the contract operates without substantial rounding errors.

Go back to Findings Summary

M1: renounceOwnership locks token

Medium severity issue

Impact:	High	Likelihood:	Low
Target:	DisperseBreed.sol	Type:	Access control

Description

The `DisperseBreed` contract inherits from the `Ownable` abstract contract, which provides the `renounceOwnership()` function. Typically, this function is used when ownership is needed only during the initialization phase, allowing the contract to function without an owner afterward. However, in the `DisperseBreed` contract, ownership is tied to critical operations such as token withdrawals. As a result, calling `renounceOwnership()` would unintentionally remove the owner, leaving the contract without the necessary permissions to withdraw tokens.

Exploit scenario

Bob, who owns the `DisperseBreed` contract, calls the `renounceOwnership()` function. After that, the contract no longer recognizes Bob as the owner, and any tokens sent to it would be locked indefinitely, making them frozen.

Recommendation

Overwrite the `renounceOwnership()` function and revert when function is called.

Fix 1.1

The issue was fixed by overwriting the `renounceOwnership()` function to always revert.

Go back to Findings Summary

L1: Locked ether

Low severity issue

Impact:	Medium	Likelihood:	Low
Target:	DisperseBreed.sol	Type:	Configuration

Description

The contract 'DisperseBreed.sol' has the `payable` modifiers on the `disperseToken` and on `withdraw` functions. These modifiers allow the contract to receive ether, but they are not necessary for the contract's functionality.

Listing 2. Excerpt from DisperseBreed

```
10 function disperseToken(
11     IERC20 token,
12     address[] calldata recipients,
13     uint256[] calldata amounts
14 ) external payable {
```

Listing 3. Excerpt from DisperseBreed

```
24 function withdraw(
25     IERC20 token,
26     address recipient
27 ) external payable onlyOwner {
```

This issue has been reported by Wake static analysis tool.

Exploit scenario

Alice holds 1 Ether and intends to disperse a token across three accounts.

1. Alice calls the `disperseToken` function with 1 Ether and valid parameters.
2. The function executes without reverting, resulting in Alice losing 1 Ether, which is permanently locked in the contract.

While the probability of this scenario is low, this may lead to a loss of user funds.

Recommendation

Remove the `payable` modifier from mentioned functions.

Fix 1.1

The issue was fixed by removing the `payable` modifier from the `disperseToken` and `withdraw` functions.

Go back to Findings Summary

L2: Fixed conversion rate arbitrage

Low severity issue

Impact:	Medium	Likelihood:	Low
Target:	token-migrator.sol	Type:	Logic error

Description

The `TokenMigrator` contract provides a feature to exchange Source Token for Destination Token with a static price.

There can be a possibility of arbitrage opportunity when the Destination Token on the market is higher than the Source Token price on top of the conversion rate.

Exploit scenario

The `TokenMigrator` contract has conversion rate 1:1 for the Source Token and Destination Token.

Alice found out that Destionation Token has higher price on the market than the Source Token. Alice bought 100 Source Token and migrated to 100 Destination Token taking advantage of the arbitrage opportunity.

Recommendation

Monitor arbitrage opportunities closely and disable the migration when necessary to prevent any negative impact on the project.

Acknowledgment 1.1

The client acknowledged the issue with the following comment: ”

> *We expect that the market can function well enough to gap*

> *slippage between pricing and in case of a major fluctuation we'll setup alerts and we'll be ready to pause functionality.*
>
> — BreederDAO Team

Go back to Findings Summary

L3: Two step ownership is not used

Low severity issue

Impact:	Medium	Likelihood:	Low
Target:	DisperseBreed.sol	Type:	Access control

Description

The `DisperseBreed` contract currently uses the `transferOwnership` function to transfer ownership by directly setting the new owner's address. This approach can lead to potential risks if the wrong address is mistakenly provided, as there is no intermediate confirmation step to verify ownership transfer.

Exploit scenario

Bob, who is the admin of the `DisperseBreed` contract, mistakenly transferred ownership to an incorrect address. After that, the contract becomes inaccessible, potentially locking critical functionality or assets within the system, including tokens.

Recommendation

Implement the `Ownable2Step` abstract contract, which introduces a two-step ownership transfer process. This adds a confirmation step to ensure the ownership transfer is intentional and correct, minimizing the risk of accidental misconfiguration.

Acknowledgment 1.1

The issue was acknowledged by the client.

Go back to Findings Summary

W1: Event emits hash

Impact:	Warning	Likelihood:	N/A
Target:	DisperseBreed.sol	Type:	Data validation

Description

The event with indexed array argument returns hash of the array. As a result, it does not return actual array of elements.

Listing 4. Excerpt from DisperseBreed

```
8 event Disperse(address[] indexed recipients, uint256[] indexed amounts);
```

Recommendation

There are two ways to remediate this issue:

- `indexed` can be removed and the array is correctly emitted,
- modified event `Disperse(recipient, amounts)` is emmited in every iteration of the loop.

Fix 1.1

The issue was fixed by removing the `indexed` for the event argument. The `event Disperse(address[] recipients, uint256[] amounts);` is used.

Go back to Findings Summary

W2: Burn is not used during token migration

Impact:	Warning	Likelihood:	N/A
Target:	token-migrator.sol	Type:	Logic error

Description

In the `migrate` function, the source token will be sent to `0xdead`. There is the `burn` function that can be used instead and decrease `totalSupply` of token.

Recommendation

Give the `TokenMigrator` contract the `BURNER_ROLE` role to burn the token and decrease `totalSupply` of token.

Acknowledgment 1.1

The client acknowledged the issue with the following comment:

> *Burn role has been relinquished together with admin and we cant assign possible burners, 0xdead will be our best option.*
>
> — BreederDAO Team

Go back to Findings Summary

W3: Unused safeERC20

Impact:	Warning	Likelihood:	N/A
Target:	DisperseBreed.sol	Type:	Code quality

Description

The `DisperseBreed` contract currently does not use the `SafeERC20` library when interacting with ERC-20 tokens. While the contract is designed to work with specific tokens such as `BreederToken` and `SovrunToken`, it does not restrict its usage exclusively to these tokens.

Listing 5. Excerpt from DisperseBreed

```
17 for (uint256 i = 0; i < recipients.length; i++) {
18     require(token.transferFrom(msg.sender, recipients[i], amounts[i]));
19 }
```

Listing 6. Excerpt from DisperseBreed

```
30 require(token.transfer(recipient, tokenBalance), "transfer failed");
```

This omission introduces a potential risk if the contract interacts with other tokens. Some ERC-20 tokens may return `true` even if the transfer fails, causing silent transaction failures. Without `SafeERC20`, the contract would not properly handle such edge cases, leading to unexpected behavior or loss of funds during token transfers.

Recommendation

To improve code safety and reliability, replace all direct token transfer calls with `SafeERC20` methods. This ensures that token operations such as `transfer`, `transferFrom`, and `approve` correctly revert the transaction if they fail, preventing silent failures.

Fix 1.1

The issue was fixed by using the safeERC20 library.

Go back to Findings Summary

I1: Check approval once to save gas

Impact:	Info	Likelihood:	N/A
Target:	DisperseBreed.sol	Type:	Gas optimization

Description

The function `disperseToken` checks the approval of the token for the DisperseBreed contract every time as `.transferFrom(…)` is called in the loop.

Listing 7. Excerpt from DisperseBreed

```
17 for (uint256 i = 0; i < recipients.length; i++) {
18     require(token.transferFrom(msg.sender, recipients[i], amounts[i]));
19 }
```

Recommendation

Implement the check of the approval only once in the following way:

```
function disperseToken(
    IERC20 token,
    address[] calldata recipients,
    uint256[] calldata amounts
) external {
    require(recipients.length == amounts.length, "not equal");

    uint256 totalValue = 0;
    for (uint256 i = 0; i < recipients.length; ++i) {
        totalValue += amounts[i];
    }
    require(token.transferFrom(msg.sender, address(this), totalValue));

    for (uint256 i = 0; i < recipients.length; i++) {
        require(token.transfer(recipients[i], amounts[i]));
    }
    // <--SNIP-->
}
```

A complete measurement and comparison of gas consumption between the two approaches utilizing the Wake framework are to be found in Appendix A.

Fix 1.1

The issue was fixed by implementing a check of the approval only once and, after the check, transferring tokens to the smart contract, which disperses them to recipients.

Go back to Findings Summary

I2: Natspec is missing

Impact:	Info	Likelihood:	N/A
Target:	DisperseBreed.sol	Type:	Code quality

Description

NatSpec documentation comments are missing in the DisperseBreed.sol contract, which can help developers understand the code.

In contrast, including NatSpec comments in the `SovrunToken.sol` contract may not provide benefits due to potential overhead.

Recommendation

Add NatSpec documentation to the DisperseBreed.sol contract.

Fix 1.1

The issue was fixed by adding NatSpec documentation to the `DisperseBreed.sol` contract.

Go back to Findings Summary

I3: Inconsistent filename

Impact:	Info	Likelihood:	N/A
Target:	token-migrator.sol	Type:	Code quality

Description

Most contracts name are the Pascal case, but only `token-migrator.sol` is Kebab Case.

Recommendation

Change filename to `TokenMigrator.sol`.

Fix 1.1

The issue was fixed by changing the file name to `TokenMigrator.sol`.

Go back to Findings Summary

I4: String error used instead of custom error

Impact:	Info	Likelihood:	N/A
Target:	SovrunToken.sol	Type:	Gas optimization

Description

Using the custom error instead of the error string in combination with the `if` statement is more gas efficient.

Listing 8. Excerpt from DisperseBreed

```
15 require(recipients.length == amounts.length, "not equal");
```

It is especially useful here because it can specify which `transferFrom` caused a revert in the custom error, making debugging easier.

Listing 9. Excerpt from DisperseBreed

```
17 for (uint256 i = 0; i < recipients.length; i++) {
18     require(token.transferFrom(msg.sender, recipients[i], amounts[i]));
19 }
```

Listing 10. Excerpt from DisperseBreed

```
29 require(tokenBalance > 0, "no token balance");
30 require(token.transfer(recipient, tokenBalance), "transfer failed");
```

Recommendation

Replace `require()` with the `if` statement in combination with a custom error.

Fix 1.1

The issue was fixed by replacing the string error with the `EmptyOrMismatchedArrays()` error for reverting.

Go back to Findings Summary

Report Revision 1.1

Revision Team

Revision team is the same as in Report Revision 1.0.

Appendix A: Gas costs comparison

The following Wake test demonstrates a comparison of gas cost between the original implementation and the optimized one.

Optimized implementation is described in I1 finding.

```
from wake.testing import *

from pytypes.contracts.dispersebreed.DisperseBreed import DisperseBreed
from pytypes.contracts.dispersebreed.DisperseBreedOptimized import
DisperseBreedOptimized
from pytypes.contracts.sovruntoken.SovrunToken import SovrunToken


@default_chain.connect()
def test_default():

    owner = chain.accounts[0]

    disperse_breed_1 = DisperseBreed.deploy()
    disperse_breed_2 = DisperseBreedOptimized.deploy()

    sovrun_token = SovrunToken.deploy()
    sovrun_token.grantRole(sovrun_token.MINTER_ROLE(), owner, from_=owner)

    user1 = chain.accounts[1]
    user2 = chain.accounts[2]
    user3 = chain.accounts[3]
    user4 = chain.accounts[4]
    user5 = chain.accounts[5]
    user6 = chain.accounts[6]
    user7 = chain.accounts[7]
    user8 = chain.accounts[8]

    sovrun_token.mint(owner, uint256(3200*10**18), from_=owner)

    # mint tokens also to users to them not have zero value in storage
    sovrun_token.mint(user1, uint256(1*10**18), from_=owner)
    sovrun_token.mint(user2, uint256(1*10**18), from_=owner)
    sovrun_token.mint(user3, uint256(1*10**18), from_=owner)
    sovrun_token.mint(user4, uint256(1*10**18), from_=owner)
    sovrun_token.mint(user5, uint256(1*10**18), from_=owner)
```

```
    sovrun_token.mint(user6, uint256(1*10**18), from_=owner)
    sovrun_token.mint(user7, uint256(1*10**18), from_=owner)
    sovrun_token.mint(user8, uint256(1*10**18), from_=owner)

    sovrun_token.approve(disperse_breed_1, uint256(1600*10**18), from_=owner)
    sovrun_token.approve(disperse_breed_2, uint256(1600*10**18), from_=owner)

    tx_1 = disperse_breed_1.disperseToken(sovrun_token,
        [user1, user2, user3, user4, user5, user6, user7, user8],
        [
            uint256(200*10**18), uint256(200*10**18), uint256(200*10**18),
            uint256(200*10**18), uint256(200*10**18), uint256(200*10**18),
            uint256(200*10**18), uint256(200*10**18)
        ],
        from_=owner)
    tx_2 = disperse_breed_2.disperseToken(sovrun_token,
        [user1, user2, user3, user4, user5, user6, user7, user8],
        [
            uint256(200*10**18), uint256(200*10**18), uint256(200*10**18),
            uint256(200*10**18), uint256(200*10**18), uint256(200*10**18),
            uint256(200*10**18), uint256(200*10**18)
        ],
        from_=owner)

    print("gas used by first disperse: ", tx_1.gas_used)
    print("gas used by second disperse: ", tx_2.gas_used)
```

The result of the test is:

```
tests/test_gas_opt.py Launching anvil --prune-history 100 --transaction-block
-keeper 10 --steps-tracing --silent --port 51360
gas used by first disperse:  152022
gas used by second disperse:  138030
```

Where implementation of DisperseBreedOptimized is:

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.23;

import "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import "@openzeppelin/contracts/access/Ownable.sol";
```

```
contract DisperseBreedOptimized is Ownable {
    event Disperse(address[] indexed recipients, uint256[] indexed amounts);

    function disperseToken(
        IERC20 token,
        address[] calldata recipients,
        uint256[] calldata amounts
    ) external payable {
        require(recipients.length == amounts.length, "not equal");

        uint256 totalValue = 0;
        for (uint256 i = 0; i < amounts.length; ++i) {
            totalValue += amounts[i];
        }

        require(token.transferFrom(msg.sender, address(this), totalValue));

        for (uint256 i = 0; i < recipients.length; i++) {
            require(token.transfer(recipients[i], amounts[i]));
        }

        emit Disperse(recipients, amounts);
    }

    function withdraw(
        IERC20 token,
        address recipient
    ) external payable onlyOwner { // why is it even payable also?
        uint256 tokenBalance = token.balanceOf(address(this));
        require(tokenBalance > 0, "no token balance");
        require(token.transfer(recipient, tokenBalance), "transfer failed");
    }
}
```

Appendix B: How to cite

Please cite this document as:

> Ackee Blockchain Security, BreederDAO: Token migrator, 22.11.2024.

Appendix C: Wake Findings

This section lists the outputs from the Wake framework used for testing and static analysis during the audit.

C.1. Fuzzing

The following table lists all implemented execution flows in the Wake fuzzing framework.

ID	Flow	Added
F1	Transfer Breeder Token	1.0
F2	Disperse Breeder Token	1.0
F3	Migrate Breeder Token	1.0
F4	Enable Migration	1.0
F5	Disable Migration	1.0
F6	Mint Subrun Token	1.0
F7	Burn Subrun Token	1.0
F8	Transfer Subrun Token	1.0

Table 4. Wake fuzzing flows

The following table lists the invariants checked after each flow.

ID	Invariant	Added	Status
IV1	Transactions do not revert except where explicitly expected	1.0	Success
IV2	Enabling/Disabling token migration works as expected	1.0	Success
IV3	Account balancing of Breeder/Sovrun is matching expected values	1.0	Success

ID	Invariant	Added	Status
IV4	TotalSupply of Subrun/Breeder tokens equals to expected	1.0	**Fail** (W2)

Table 5. Wake fuzzing invariants

C.2. Detectors

```
wake detect locked-ether

[HIGH][MEDIUM] Contract receives ether but never sends it. [locked-ether]
   4 import "@openzeppelin/contracts/token/ERC20/IERC20.sol";
   5 import "@openzeppelin/contracts/access/Ownable.sol";
   6
 ❱ 7 contract DisperseBreed is Ownable {
   8     event Disperse(address[] indexed recipients, uint256[] indexed amounts);
   9
  10     function disperseToken(
contracts/disperse-breed/DisperseBreed.sol
  This function can receive ether.
  21         emit Disperse(recipients, amounts);
  22     }
  23
❱ 24     function withdraw(
  25         IERC20 token,
  26         address recipient
  27     ) external payable onlyOwner {
contracts/disperse-breed/DisperseBreed.sol
  This function can receive ether.
   7 contract DisperseBreed is Ownable {
   8     event Disperse(address[] indexed recipients, uint256[] indexed amounts);
   9
❱ 10     function disperseToken(
  11         IERC20 token,
  12         address[] calldata recipients,
  13         uint256[] calldata amounts
contracts/disperse-breed/DisperseBreed.sol
```

Figure 1. Locked ether

```
wake detect unsafe-erc20-call

[MEDIUM][HIGH] Unsafe variant of ERC-20 call [unsafe-erc20-call]
 15         require(recipients.length == amounts.length, "not equal");
 16
 17         for (uint256 i = 0; i < recipients.length; i++) {
 18             require(token.transferFrom(msg.sender, recipients[i], amounts[i]));
 19         }
 20
 21         emit Disperse(recipients, amounts);
contracts/disperse-breed/DisperseBreed.sol

[MEDIUM][HIGH] Unsafe variant of ERC-20 call [unsafe-erc20-call]
 27     ) external payable onlyOwner {
 28         uint256 tokenBalance = token.balanceOf(address(this));
 29         require(tokenBalance > 0, "no token balance");
 30         require(token.transfer(recipient, tokenBalance), "transfer failed");
 31     }
 32 }
contracts/disperse-breed/DisperseBreed.sol
```

Figure 2. Unsafe ERC-20 call

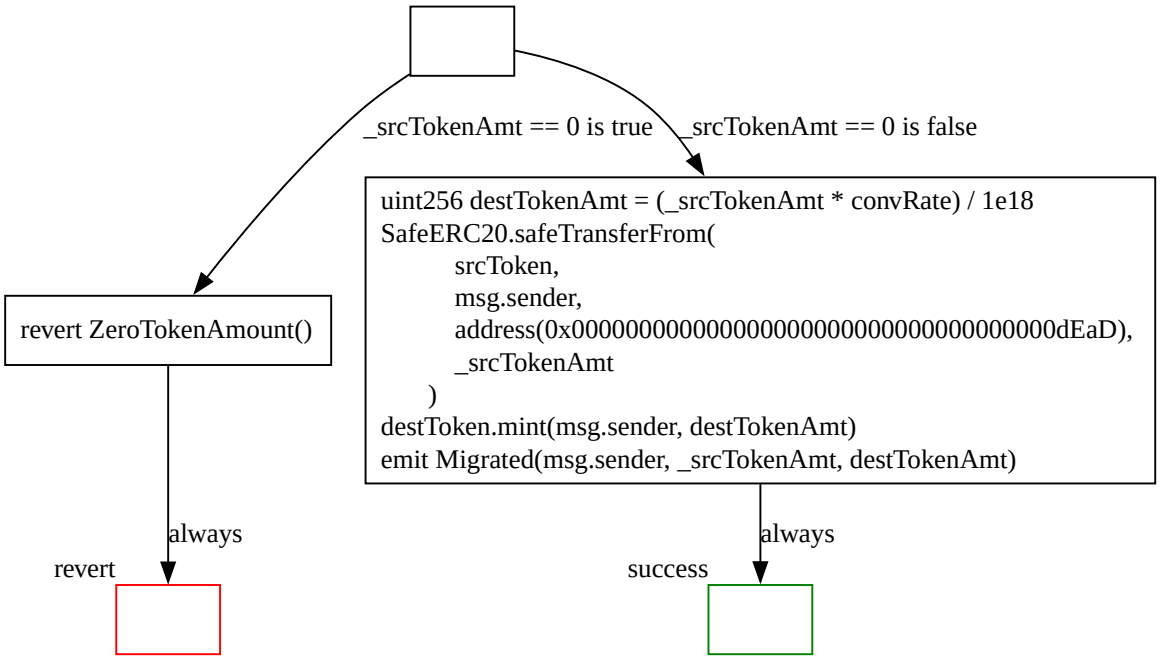


Figure 3. Control Flow Graph of migrate function

ackee
blockchain security

Thank You

Ackee Blockchain a.s.

Rohanske nabrezi 717/4
186 00 Prague
Czech Republic

hello@ackee.xyz