ackee blockchain security

Stable Labs

Token & Treasury contracts

by Ackee Blockchain

19.7.2024

Contents

1. Document Revisions

1.0-draft	Draft report	1.7.2024
1.0	Final report	1.7.2024
1.1	Fix review	19.7.2024

2. Overview

This document presents our findings in reviewed contracts.

2.1. Ackee Blockchain

Ackee Blockchain is an auditing company based in Prague, Czech Republic, specializing in audits and security assessments. Our mission is to build a stronger blockchain community by sharing knowledge – we run free certification courses School of Solana, Summer School of Solidity and teach at the Czech Technical University in Prague. Ackee Blockchain is backed by the largest VC fund focused on blockchain and DeFi in Europe, RockawayX.

2.2. Audit Methodology

1. **Technical specification/documentation** - a brief overview of the system is requested from the client and the scope of the audit is defined.
2. **Tool-based analysis** - deep check with automated Solidity analysis tools and Wake is performed.
3. **Manual code review** - the code is checked line by line for common vulnerabilities, code duplication, best practices and the code architecture is reviewed.
4. **Local deployment + hacking** - the contracts are deployed locally and we try to attack the system and break it.
5. **Unit and fuzz testing** - run unit tests to ensure that the system works as expected, potentially write missing unit or fuzz tests.

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 (such as deployment scripts, compiler configuration, use of 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**	**-**
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 (such as deployment scripts, compiler configuration, use of multi-signature wallets for owners, etc.), 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 (see above) 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

Member's Name	Position
Jan Kalivoda	Lead Auditor
Dima Khimchenko	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

Stable Labs is a project that introduces a new stablecoin.

Revision 1.0

Stable Labs engaged Ackee Blockchain to perform a security review of the Stable Labs protocol with a total time donation of 5 engineering days in a period between June 24 and June 28, 2024, with Jan Kalivoda as the lead auditor.

The audit was performed on the commit `79d08d4` [1] and the scope was the following:

- src/connectorLayer/TreasuryOrchestrator.sol
- src/tokens/StRWA.sol
- src/tokens/StStable.sol
- src/utils/Greenlist.sol
- src/utils/Treasury.sol

We began our review using static analysis tools, including Wake. We then took a deep dive into the logic of the contracts. For testing and fuzzing, we have involved Wake testing framework. 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,
- looking for common issues such as data validation.

Our review resulted in 16 findings, ranging from Info to High severity. The most severe issues are caused by insufficient testing and can be discovered by only executing the functions.

Ackee Blockchain recommends Stable Labs:

- write a comprehensive test suite, ideally including fuzz tests,
- address all reported issues.

See Revision 1.0 for the system overview of the codebase.

Revision 1.1

The fix review was done on the given commit: `5750567` [2] and the scope were the fixes of the issues from Revision 1.0.

All the issues were addressed and fixed, except the following:

- I1: Code duplication
- I4: The `encodedReleases` mapping is not used
- I8: Inefficient array iterations

See the updated findings table for the updated status of all the issues.

[1] full commit hash: `79d08d43d633e4573f5f1a435edc44215b7f9dcb`

[2] full commit hash: `5750567aa4f7b32c59cdba143438fff12c17f9e9`

4. Summary of Findings

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

- a *Description*,
- an *Exploit scenario*,
- a *Recommendation* and if applicable
- a *Fix*.

There might often be multiple ways to solve or alleviate the issue, with varying requirements regarding the necessary changes to the codebase. In that case, we will try to enumerate them all, clarifying which solves the underlying issue better (albeit possibly only with architectural changes) than others.

	Severity	Reported	Status
H1: Wipe logic does not work	High	1.0	Fixed
H2: Locked tokens due to missing approval	High	1.0	Fixed
M1: Renounce ownership	Medium	1.0	Fixed
L1: Revert inconsistency on transfer	Low	1.0	Fixed
L2: Double-entrypoint for the `initialize` function	Low	1.0	Fixed
L3: Missing events	Low	1.0	Fixed
W1: Inconsistent usage of `msg.sender` and `_msgSender()`	Warning	1.0	Fixed

	Severity	Reported	Status
W2: Potential storage clashes	Warning	1.0	Fixed
I1: Code duplication	Info	1.0	Acknowledged
I2: Unused import	Info	1.0	Fixed
I3: Unused event	Info	1.0	Fixed
I4: The `encodedReleases` mapping is not used	Info	1.0	Acknowledged
I5: The release functions are similar	Info	1.0	Fixed
I6: Ambiguous naming of a function	Info	1.0	Fixed
I7: Inconsistent usage of modifiers and checks in function's body	Info	1.0	Fixed
I8: Inefficient array iterations	Info	1.0	Acknowledged

Table 2. Table of Findings

5. Report revision 1.0

5.1. System Overview

This section contains an outline of the audited contracts. Note that this is meant for understandability purposes and does not replace project documentation.

Contracts

Contracts we find important for better understanding are described in the following section.

StStable

The ERC20 token that is upgradeable, permit enabled, pausable, ownable, and allows to blacklist addresses.

StRWA

The ERC20 token that is upgradeable, permit enabled, pausable, ownable, and allows to blacklist addresses.

Greenlist

The contract allows to set addresses on greenlist by the owner.

Treasury

The contract that holds and manages tokens. The tokens can be released by authorized addresses or the owner.

TreasuryOrchestrator

The contract allows to manage multiple treasuries and do operations over them.

Actors

This part describes actors of the system, their roles, and permissions.

Owner

The owner of the contract. The owner can set other roles and pause the contract.

Asset Protector

The role that can freeze addresses and wipe their balances.

Supply Controller

The role can mint and burn tokens.

Mintable Address

The address that can be the receiver of the minted tokens.

5.2. Trust Model

Users have to trust Supply Controller to not maliciously mint tokens. Also, users have to trust Asset Protector to not freeze their addresses and wipe their balances. And lastly, users have to trust Owner to set all the elevated privileges correctly and that he will not misuse the protocol.

H1: Wipe logic does not work

High severity issue

Impact:	Medium	Likelihood:	High
Target:	StRWA.sol, StStable.sol	Type:	Logic error

Description

The function for wiping address balances requires the address to be frozen.

```
function wipeFrozenAddress(address account) public onlyAssetProtector {
    if (isFrozen[account] == false) {
        revert NotFrozen();
    }
    uint256 balance = balanceOf(account);
    _burn(account, balanceOf(account));
    emit Wiped(account, balance);
}
```

However, when the `_burn` function is called, there is also called the `_update` function.

```
function _burn(address account, uint256 value) internal {
    if (account == address(0)) {
        revert ERC20InvalidSender(address(0));
    }
    _update(account, address(0), value);
}
```

That requires the address to **not** be frozen. As a result, there is a contradiction for the frozen state and can never be executed.

Exploit scenario

The `wipeFrozenAddress` function is called and the transaction always reverts.

Recommendation

Remove the second blocking constraint if it is desired to be able to wipe frozen addresses. Otherwise, remove the wipe function.

Fix 1.1

The issue was fixed by calling the parent update function directly.

```
super._update(account, address(0), balanceOf(account));
```

Go back to Findings Summary

H2: Locked tokens due to missing approval

High severity issue

Impact:	Medium	Likelihood:	High
Target:	TreasuryOrchestrator.sol	Type:	Logic error

Description

The `TreasuryOrchestrator` is supposed to hold tokens and call the `redeem` function on treasuries to send tokens from the orchestrator to the treasuries.

```
function redeem(address _token, uint256 _amount) external whenNotPaused {
    ITreasury(treasuryContracts[_token]).redeem(_amount);
}
```

The redeem function is calling a simple `transferFrom` function.

```
function redeem(uint256 amount) external whenNotPaused {
    token.safeTransferFrom(msg.sender, address(this), amount);
    emit Redeemed(msg.sender, amount);
}
```

However, the `TreasuryOrchestrator` contract does not set approval anywhere. As a result, the tokens are locked in the contract. Since the contract is upgradable, they can be unlocked with a new implementation.

Exploit scenario

The `redeem` function is called and reverts.

Recommendation

Implement the token approvals.

Fix 1.1

The `redeem` function was removed. The contract is not supposed to hold any tokens.

[Go back to Findings Summary]

M1: Renounce ownership

Medium severity issue

Impact:	High	Likelihood:	Low
Target:	`**/*`	Type:	Data validation

Description

The contracts inherit from Openzeppelin's `Ownable2StepUpgradeable` contract. Due to that, the ownership can be by default renounced by the owner. For the Treasury contracts, it is not desirable to lose access to manage them. Also, it might not be desirable for the tokens.

Exploit scenario

The owner accidentally calls `renounceOwnership()`. Then the owner is lost.

Recommendation

Override the `renounceOwnership()` method to disable this feature if it is not intended to be used in the future (wherever it is needed).

Fix 1.1

The renounce ownership function is using two-step pattern.

[Go back to Findings Summary]

L1: Revert inconsistency on transfer

Low severity issue

Impact:	Low	Likelihood:	Medium
Target:	StRWA.sol, StStable.sol	Type:	Best practices

Description

The frozen addresses can't participate in transfer, burn or mint operations, because of the overridden `_update` function that is called as a hook for these operations.

```
function _update(
    address from,
    address to,
    uint256 amount
) internal virtual override whenNotPaused {
    if (isFrozen[from] || isFrozen[to]) {
        revert AddressFrozen();
    }
    super._update(from, to, amount);
}
```

However, the transfer function is also overridden with the same requirement but a different error.

```
function transfer(
    address to,
    uint256 amount
) public virtual override(ERC20Upgradeable) returns (bool) {
    if (isFrozen[msg.sender] || isFrozen[to]) {
        revert FrozenAddressIncludedInTransfer();
    }
    super.transfer(to, amount);
    return true;
}
```

The `transferFrom` function is not overridden for both contracts. As a result, we have 3 different errors for the same requirement.

- The `AddressFrozen` error - for `transferFrom` in StRWA,
- the `FrozenAddressIncludedInTransfer` error - for `transfer` in StStable and StRWA,
- the `ContractFrozen` error - for `transferFrom` in StStable.

This cannot cause problems in the current scope, but it can cause more serious problems with future development or on the off-chain side.

Exploit scenario

The `AddressFrozen` error is expected, instead of the `FrozenAddressIncludedInTransfer` error is provided.

Recommendation

Remove the inconsistency, for example, by using only the `_update` function and one error type.

Fix 1.1

The revert inconsistency was removed.

Go back to Findings Summary

L2: Double-entrypoint for the `initialize` function

Low severity issue

Impact:	Low	Likelihood:	Low
Target:	Treasury.sol, Greenlist.sol	Type:	Logic error

Description

The `Treasury` contract has the following `initialize` function.

```
function initialize(address _owner, IERC20 _token) public initializer {
    _transferOwnership(_owner);
    token = _token;
}
```

However, it has also the function inherited from the `Greenlist` contract.

```
function initialize(address _owner) public initializer {
    _transferOwnership(_owner);
}
```

As a result, it is possible to initialize `Treasury` without setting the token.

Exploit scenario

The `Treasury` contract is deployed and initialized without specifying the token, so it needs redeployment.

Recommendation

Only the `initialize` function from the `Treasury` contract should be present.

Fix 1.1

The initialize function from the `Greenlist` contract was removed.

Go back to Findings Summary

L3: Missing events

Low severity issue

Impact:	Low	Likelihood:	Medium
Target:	Treasury.sol, TreasuryOrchestrator.sol	Type:	Logic error

Description

The contracts are missing several events. The following list recommends functions where an event should be added.

- `TreasuryOrchestrator.addBatchTreasurycontracts` (exists for a single operation)
- `TreasuryOrchestrator.removeBatchTreasuryContracts` (exists for a single operation)
- `TreasuryOrchestrator.switchAuthorized` (critical state change)
- `Treasury.switchAuthorized` (critical state change)
- `Treasury.createReleaseRequest` (critical state change)
- `Treasury.withdraw` (critical state change)

Exploit scenario

The event is not emitted. As a result, the off-chain system that relies on it fails.

Recommendation

Add the missing events.

Fix 1.1

The events were added.

[Go back to Findings Summary]

W1: Inconsistent usage of `msg.sender` and `_msgSender()`

Impact:	Warning	Likelihood:	N/A
Target:	`**/*`	Type:	Logic error

Description

The codebase is using inconsistently `msg.sender` and `_msgSender()`. In the current codebase it not an issue since it is returning the same value, however if the logic of `_msgSender()` changes in the future, it can lead to an unexpected behavior. For example, the authorization is checked against the `msg.sender`.

```
function _checkAuthorized() internal view {
    if (!authorized[msg.sender]) {
        revert NotAuthorized();
    }
}
```

But the owner is checked against `_msgSender()`.

```
function _checkOwner() internal view virtual {
    if (owner() != _msgSender()) {
        revert OwnableUnauthorizedAccount(_msgSender());
    }
}
```

Recommendation

Replace `msg.sender` occurrences with `_msgSender()` across the whole codebase.

Fix 1.1

The `msg.sender` occurences were replaced with `_msgSender()`.

[Go back to Findings Summary]

W2: Potential storage clashes

Impact:	Warning	Likelihood:	N/A
Target:	Treasury.sol, Greenlist.sol	Type:	Logic error

Description

The codebase mostly inherits from contracts with unstructured storage. However, the `Treasury` contract inherits from `Greenlist` which has declared some storage variables and has no storage gaps. Changes to this contract can cause storage clashes in the `Treasury` contract.

Recommendation

Implement storage gaps for the `Greenlist` contract or mention in the developers' documentation that new storage variables should not be added.

Fix 1.1

The storage gaps were implemented in the `Greenlist` contract.

[Go back to Findings Summary](#)

I1: Code duplication

Impact:	Info	Likelihood:	N/A
Target:	StStable.sol, StRWA.sol	Type:	Best practices

Description

The `StStable` and `StRWA` contracts share a lot of code. To reduce code duplication and easily see differences between tokens, they could inherit from the same base contract.

Recommendation

Introduce a base contract for `StStable` and `StRWA` to inherit from.

[Go back to Findings Summary]

I2: Unused import

Impact:	Info	Likelihood:	N/A
Target:	StRWA.sol	Type:	Best practices

Description

The `StRWA` contract has the following unused import.

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

And the `TreasuryOrchestrator` contract has the following unused import.

```
import "../Interfaces/IGreenlist.sol";
```

Recommendation

Remove the unused imports.

Fix 1.1

The unused import from `StRWA` was removed. The unused import from `TreasuryOrchestrator` could be also removed.

Go back to Findings Summary

I3: Unused event

Impact:	Info	Likelihood:	N/A
Target:	TreasuryOrchestrator.sol	Type:	Best practices

Description

The `TreasuryOrchestrator` contract has the following unused events.

- `TokensReleased`
- `GreenlistedTokensReleased`

Recommendation

Remove the unused events.

Fix 1.1

The unused events were removed.

[Go back to Findings Summary](#)

I4: The `encodedReleases` mapping is not used

Impact:	Info	Likelihood:	N/A
Target:	Treasury.sol	Type:	Best practices

Description

The `Treasury` contract has functions to release tokens and a function to create a release request.

```
function createReleaseRequest(
    address _to,
    uint256 _amount
) external nonZeroAddress(_to) whenNotPaused returns (bytes memory) {
    _checkAuthorized();

    if (_amount == 0) {
        revert ZeroAmount();
    }

    encodedReleases[_to] = abi.encode(_to, _amount);

    return encodedReleases[_to];
}
```

However, these release functions do not use the created release requests. The `encodedReleases` mapping is not used anywhere in the code. As a result, it is not anyhow ensured that the mapping matches the past or future release requests.

Recommendation

Remove the function and mapping or utilize it (emit an event, check against the mapping in the release functions, etc.).

Go back to Findings Summary

I5: The release functions are similar

Impact:	Info	Likelihood:	N/A
Target:	Treasury.sol	Type:	Best practices

Description

The `Treasury` contract has the following two functions to release tokens.

```
function releaseTokens(bytes memory data) external whenNotPaused {
    _checkAuthorized();

    (address to, uint256 amount) = abi.decode(data, (address, uint256));
    if (token.balanceOf(address(this)) < amount) {
        revert NotEnoughTokens();
    }

    token.safeTransfer(to, amount);

    emit Released(to, amount);
}
```

```
function greenListRelease(
    address _to,
    uint256 _amount
) external isGreenListed(_to) whenNotPaused {
    _checkAuthorized();

    if (token.balanceOf(address(this)) < _amount) {
        revert NotEnoughTokens();
    }

    token.safeTransfer(_to, _amount);

    emit GreenListRelease(_to, _amount);
}
```

These functions do the same with the difference that the `greenListRelease` function has the `isGreenListed` modifier. So, it has additional restrictions. However, since it is doing the same then the authorized address does not need to call the `greenListRelease` function at all.

Recommendation

Reasonably adjust the logic.

Fix 1.1

The greenlist release function was removed.

Go back to Findings Summary

I6: Ambiguous naming of a function

Impact:	Info	Likelihood:	N/A
Target:	Treasury.sol	Type:	Best practices

Description

The `addTreasuryContract` function (and `addBatchTreasuryContracts` respectively) has ambiguous naming since it sets the address, not adds. It is possible to overwrite an existing treasury address by calling the `addTreasuryContract` function twice.

```
function addTreasuryContract(
    address _token,
    address _treasury
) external nonZeroAddress(_token) nonZeroAddress(_treasury) whenNotPaused {
    _checkOwner();
    treasuryContracts[_token] = _treasury;
    emit TreasuryContractAdded(_token, _treasury);
}
```

Recommendation

Adjust the function's naming or add protection to overwrite the existing treasury address by repeatedly calling the `addTreasuryContract` function.

Fix 1.1

The function was renamed to `setTreasuryContract` (and `setBatchTreasuryContracts` respectively).

[Go back to Findings Summary]

I7: Inconsistent usage of modifiers and checks in function's body

Impact:	Info	Likelihood:	N/A
Target:	`**/*`	Type:	Best practices

Description

There is inconsistent usage of in-function and modifier checks for the sender address in the codebase. It is not a security issue; adhering to one approach consistently is considered best practice. Below are two functions from `Treasury.sol` that demonstrate differing techniques for sender rights verification:

```
function switchAuthorized(
    address _address
) external nonZeroAddress(_address) whenNotPaused {
    _checkOwner();
    authorized[_address] = !authorized[_address];
}
```

```
function _checkOwner() internal view virtual {
    if (owner() != _msgSender()) {
        revert OwnableUnauthorizedAccount(_msgSender());
    }
}
```

The above examples highlight just one of the multiple similar occurrences found throughout the codebase.

Recommendation

Choose either modifiers or functions approach and use it consistently across the codebase

Fix 1.1

The modifiers are now used consistently throughout the codebase.

[Go back to Findings Summary]

I8: Inefficient array iterations

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

Description

The `TreasuryOrchestrator` contract has a modifier to check against zero addresses.

```
modifier nonZeroAddressBatch(address[] memory _address) {
    uint addressLength = _address.length;
    for (uint i = 0; i < addressLength; i++) {
        if (_address[i] == address(0)) {
            revert ZeroAddress();
        }
    }
    _;
}
```

That is used for batch operations.

```
function addBatchTreasuryContracts(
    address[] memory _tokens,
    address[] memory _treasuries
)
    external
    nonZeroAddressBatch(_tokens)
    nonZeroAddressBatch(_treasuries)
    whenNotPaused
{
```

This is inefficient because the function is already iterating through the elements, and with this modifier, it iterates twice before even stepping the function. The validation can be performed during the main iteration.

Recommendation

Perform the validation directly in the main loop in the function.

Go back to Findings Summary

Appendix A: How to cite

Please cite this document as:

Ackee Blockchain, Stable Labs: Token & Treasury contracts, 19.7.2024.

ackee | blockchain security

Appendix B: Glossary of terms

The following terms might be used throughout the document:

Superclass/Ancestor of C

A contract that C inherits/derives from.

Subclass/Child of C

A contract that inherits/derives from C.

Syntactic contract

A Solidity contract. May have an inheritance chain, and may be deployed.

Deployed contract

An EVM account with non-zero code. If its source was written in Solidity, it was created through at least one syntactic contract. If that contract had superclasses (parents), it would be composed of multiple syntactic contracts.

Init/initialization function

A non-constructor function that serves as an initializer. Often used in upgradeable contracts.

External entrypoint

A `public` or `external` function.

Public/Publicly-accessible function/entrypoint

An `external` or `public` function that can be successfully executed by any network account.

Mutating function

A non-`view` and non-`pure` function.

Thank You

Ackee Blockchain a.s.

Prague, Czech Republic

hello@ackeeblockchain.com

https://twitter.com/AckeeBlockchain