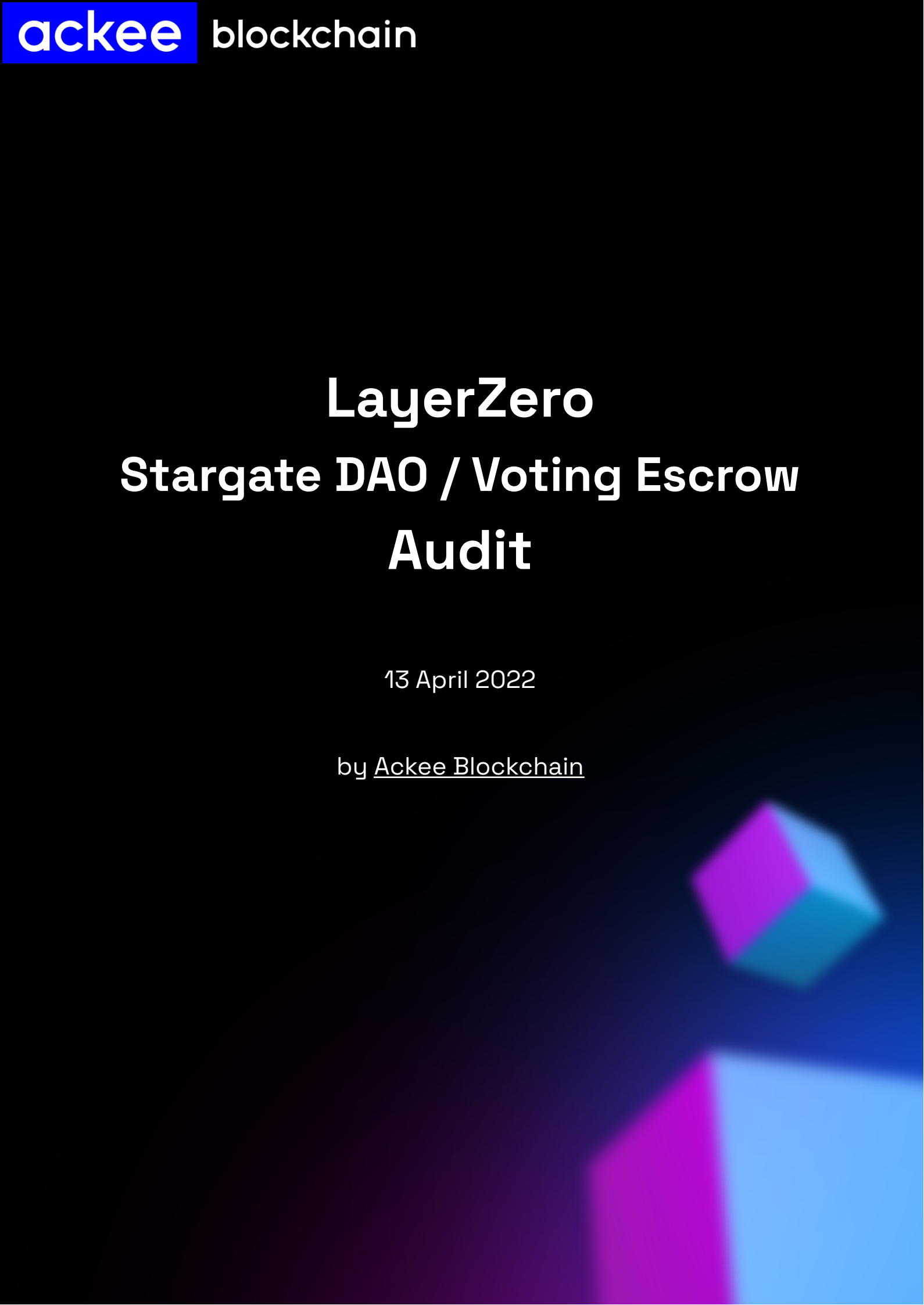
ackee blockchain

LayerZero
Stargate DAO / Voting Escrow
Audit

13 April 2022

by Ackee Blockchain



Contents

1. Document Revisions

Revision	Date	Description
1.0	29 Mar 2022	Initial revision
1.1	13 April 2022	Audit update

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, specialized in audits and security assessments. Our mission is to build a stronger blockchain community by sharing knowledge – we run a free certification course 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, Rockaway Blockchain Fund.

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 Slither is performed.
3. **Manual code review** - the code is checked line by line for common vulnerabilities, code duplication, best practices. The code architecture is reviewed.
4. **Local deployment + hacking** - contracts are deployed locally and we try to attack the system and break it.
5. **Unit testing** - run unit tests to ensure that the system works as expected. Potentially we write our own unit tests for specific suspicious scenarios.

2.3 Review team

Member's Name	Position
Štěpán Šonský	Lead Auditor
Lukáš Böhm	Auditor
Josef Gattermayer, Ph.D.	Audit Supervisor

2.4 Disclaimer

We have put our best effort to find all vulnerabilities in the system. However, our findings should not 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

LayerZero engaged Ackee Blockchain to conduct a security review of Voting Escrow protocol with a total time donation of 6 engineering days.

The scope included the following repository with a given commit:

- Private repository
  - **Rev. 1.0:** ~~ecd5793659dc2ec2dc29ad6bfeb1289963cf0258~~
  - **Rev. 1.1:** 3030c8d152a3b1d20f5f2530e33bea28121a0cac

We began our review using static analysis tools and then took a deep dive into the logic of the contracts. During the review, we paid particular attention to:

- Checking if nobody can exploit the protocol.
- Ensuring access controls are not too weak.
- Validating the math model.
- Checking the code quality and Solidity best practices.
- Looking for common issues such as data validation.

Our review resulted in 5 findings, mainly informational regarding the code quality and one warning. During typecasting, integer overflow usually leads to high impact issues. But in this case, we've identified the likelihood close to zero, so the final classification is the warning.

Rev. 1.1: The `VotingEscrow.sol` code has been slightly refactored. Functions' visibility and modifiers have been properly changed. These minor changes do not bring any new issues and do not affect the trust model and the security.

Ackee Blockchain recommends LayerZero to:

- Be aware of integer overflow during the typecasting.
- Refactor the code to avoid duplications.
- Follow the Solidity naming conventions.

4. 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.

4.1 Contracts

Both contracts are almost the exact fork of the well-known vyper contract by Curvefi. Few logic differences come from Solidity x Vyper language syntax. Both contracts properly uses `ReentrancyGuard` on external functions which change the contract state.

VotingEscrow

The contract contains two custom modifiers. Modifier `onlyUserOrWhitelist` for secure function accessibility and `notUnlocked` for handling locked/unlocked contract's state. This state is unique for `VotingEscrow`, just like whitelist management. The major part of the logic is located in the private `_checkpoint` function, where Curve's mathematical formulas calculate the voting power of a specific user which is changing during the locking period (max 3 years). A user's veSTG balance decays linearly as the remaining time until the STG unlock decreases.

Rev. 1.1: The logic of external functions `create_lock()` and `withdraw()` was moved into internal functions `_create_lock()` and `_withdraw()`. Now the original external functions just call these internal ones. This refactor was done because of the new function `withdraw_and_create_lock()`, which calls `_withdraw()` and then `_create_lock()`.

sVotingEscrow

The contract is a slightly modified version of `VotingEscrow` contract. It does not contain the `token`, so transfers and has a different trust model architecture. Instead of users and whitelisted contracts, only the contract owner has the ability to call specific functions.

4.2 Actors

Owner / LayerZero

The owner deploys contracts to the network and has extra privileges in contracts. The owner can call specific functions and modify the whitelist in case of the `VotingEscrow` contract.

VotingEscrow:

- `add_to_whitelist()`
- `remove_from_whitelist()`
- `unlock()`

sVotingEscrow:

- `create_lock_for()`
- `increase_amount_for()`
- `increase_unlock_time_for()`
- `withdraw_for()`

User
This role means any external address in the network which can interact with the protocol.

VotingEscrow:

- Read public/external contract state (user's slope, point history, locked end, balances, supply)
- `checkpoint()`
- `deposit_for()`
- `create_lock()`
- `increase_amount()`
- `increase_unlock_time()`
- `increase_amount_and_time()`
- `withdraw()`
- `withdraw_and_create_lock()`

sVotingEscrow:

- Read public/external contract state (user's slope, point history, locked end, balances, supply)

Controller
The controller role is defined in the VotingEscrow, but it's unused in the contract itself. Only the previous controller can set the new one using the `changeController()` function.

Whitelisted contract
In `VotingEscrow` contract, whitelisted contracts has the same privileges to interact with the contract as users.

4.3 Trust model

While in `sVotingEscrow` `onlyOwner` modifier is used in critical functions, `VotingEscrow` contains a white-list mapping that enables selected contracts to call specific functions. However, the contract owner can modify the whitelist. From

our perspective the trust model is well designed and the owner is not overpowered. Only whitelisted contracts have to trust the owner in terms of removing themselves from the whitelist.

5. Vulnerabilities risk methodology

Each finding contains an *Impact* and *Likelihood* ratings.

If we have found a scenario in which the issue is exploitable, it will be assigned an impact of *High*, *Medium*, or *Low*, based on the direness of the consequences it has on the system. If we have not 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 *Informational*.

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

5.1 Finding classification

The complete definitions are as follows:

Impact

High
Conditions that activate the issue will lead to undefined or catastrophic consequences for the system.

Medium
Conditions that activate the issue will result in consequences of serious substance.

Low
Conditions that activate the issue will have outcomes on the system that are either recoverable or do not 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 have not found a way to exploit the issue given the time constraints, it might be marked as "Warning" or higher, based on our best estimate of whether it is currently exploitable.

Informational
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.

6. Findings

This section contains the list of discovered findings. Unless overridden for purposes of readability, each finding contains:

- a *Description*,
- an *Exploit scenario*, and
- a *Recommendation*

Many times, there might be multiple ways to solve or alleviate the issue, with varying requirements in terms of 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. Issues can be also acknowledged by developers as not a risk.

Summary of Findings

ID		Type	Impact	Likelihood	Status
W1	Integer casting overflow	Type safety	N/A	N/A	Acknowledged
I1	Binary search code duplication	Architecture	N/A	N/A	Acknowledged
I2	Function naming convention violation	Best practices	N/A	N/A	Acknowledged
I3	Constant naming convention violation	Best practices	N/A	N/A	Acknowledged
I4	Amount signed int	Type safety	N/A	N/A	Acknowledged

W1 - Integer casting overflow

Impact:	N/A	**Likelihood:**	N/A
Target:	VotingEscrow.sol	**Type:**	Type safety

Description
Solidity >0.8 handles the integer overflow/underflow during arithmetic operations. But in case of casting, the transaction doesn't revert and value overflows.

Exploiting scenario
In the `_deposit_for()` function, we've identified this potential issue, but the probability of exploiting is very low when `token` uses standard decimals and proper total supply. However we have to point out this as a warning.

Oe line 309 if the _value (uint256) would be higher than int128.max, then the result can overflow and cause mismatch.

```
_locked.amount += int128(int(_value));
```

Then on line 322, tokens get tranferred using the original `uint256 _value`, which can be much higher than the overflown value.

```
IERC20(token).safeTransferFrom(_addr, address(this), _value);
```

Recommendation
Be aware of integer casting overflow and if this edge case is possible, add the require statement to avoid it definitely.

I1 - Binary search code duplication

Impact:	N/A	**Likelihood:**	N/A
Target:	VotingEscrow.sol	**Type:**	Architecture

Description
There is a code duplication of binary search algorithm. One is in the `_find_block_epoch()` function and another is in `balanceOfAt()` function.

Recommendation
Code duplication is generally a bad practice because it makes the code more error-prone and also becomes less readable. We recommend to make binary search more abstract and moving it into a separate function.

I2 - Function naming convention

Impact:	N/A	**Likelihood:**	N/A
Target:	VotingEscrow.sol sVotingEscrow.sol	**Type:**	Best practices

Description
The function naming is inconsistent, and function names with underscores violate Solidity naming conventions. The only justifiable case for underscores is as a prefix in internal/private functions. Inconsistent naming does not look professional from the developer's perspective.

Recommendation
We recommend using camelCase naming in all public/external functions and _camelCase in private/internal functions.

I3 - Constant naming convention

Impact:	N/A	**Likelihood:**	N/A
Target:	VotingEscrow.sol sVotingEscrow.sol	**Type:**	Best practices

Description
Some constants don't follow the Solidity naming conventions.

```
int128 internal constant iMAXTIME = 3 * 365 * 86400;
```

```
string public constant name = "veSTG";
string public constant symbol = "veSTG";
string public constant version = "1.0.0";
uint8 public constant decimals = 18;
```

Recommendation
We recommend following the naming convetions, so using the UPPER_CASE naming for constants for better readability.

I4 - Signed integer amount

Impact:	N/A	**Likelihood:**	N/A
Target:	VotingEscrow.sol sVotingEscrow.sol	**Type:**	Type safety

Description

The `amount` value in the `LockedBalance` structure is defined as a signed integer. We haven't found any reason to allow the negative `amount` value here.

Recommendation

Use the unsigned integer data type for non-negative values like the `amount`.

ackee blockchain

Thank You

Ackee Blockchain a.s.


Prague, Czech Republic


hello@ackeeblockchain.com


https://discord.gg/z4KDUbuPxq