Ekta

LinearPool

Smart Contract Audit Final Report

April 20, 2022

This audit does not provide a security or correctness guarantee of the audited smart contract. Securing smart contracts is a multistep process, therefore running a bug bounty program as a complement to this audit is strongly recommended.

Introduction

1. About Ekta

Ekta's vision is to create a world where blockchain technology is used to give everyone a chance to live a better life. A new ecosystem is needed, one where people from different backgrounds and socio-economic circumstances can participate freely, without the barriers and inefficiencies introduced by centralized governing bodies.

Ekta's mission is to bridge the blockchain world with the world we live in, and to create value in both. This is accomplished through various branches of the Ekta ecosystem, which include:

- The tokenization of real-world assets through Ekta Chain and Ekta's self-developed NFT Marketplace
- Ekta's decentralized credit platform that allows all users to participate
- Physical spaces such as the island chain currently being developed in Indonesia, where physical land and real estate assets will be brought on-chain
- Ekta's startup incubator and innovation center open to retail investment

Visit https://ekta.io/ to know more about it.

2. About ImmuneBytes

ImmuneBytes is a security start-up to provide professional services in the blockchain space. The team has hands-on experience in conducting smart contract audits, penetration testing, and security consulting. ImmuneBytes's security auditors have worked on various A-league projects and have a great understanding of DeFi projects like AAVE, Compound, 0x Protocol, Uniswap, dydx.

The team has been able to secure 105+ blockchain projects by providing security services on different frameworks. ImmuneBytes team helps start-up with a detailed analysis of the system ensuring security and managing the overall project.

Visit http://immunebytes.com/ to know more about the services.

Documentation Details

The Ekta team has provided the following doc for the purpose of audit:

1. https://whitepaper.ektaworld.io/

Audit Process & Methodology

ImmuneBytes team has performed thorough testing of the project starting with analyzing the code design patterns in which we reviewed the smart contract architecture to ensure it is structured and safe use of third-party smart contracts and libraries.

Our team then performed a formal line-by-line inspection of the Smart Contract in order to find any potential issues like Signature Replay Attacks, Unchecked External Calls, External Contract Referencing, Variable Shadowing, Race conditions, Transaction-ordering dependence, timestamp dependence, DoS attacks, and others.

In the Unit testing phase, we run unit tests written by the developer in order to verify the functions work as intended. In Automated Testing, we tested the Smart Contract with our in-house developed tools to identify vulnerabilities and security flaws.

The code was audited by a team of independent auditors which includes -

1. Testing the functionality of the Smart Contract to determine proper logic has been followed throughout.
2. Analyzing the complexity of the code by thorough, manual review of the code, line-by-line.
3. Deploying the code on testnet using multiple clients to run live tests.
4. Analyzing failure preparations to check how the Smart Contract performs in case of bugs and vulnerabilities.
5. Checking whether all the libraries used in the code are on the latest version.
6. Analyzing the security of the on-chain data.

Audit Details

- Project Name: Ekta
- Contracts Name: Linearpool.sol
- Languages: Solidity(Smart contract)
- Github commit/Smart Contract Address for audit(initial): https://github.com/ekta-dev-team/ekta_contract
- Github commit/Smart Contract Address for audit(final): https://github.com/ekta-dev-team/ekta_contract
- Platforms and Tools: Remix IDE, Truffle, Truffle Team, Ganache, Solhint, VScode, Contract Library, Slither, SmartCheck

Audit Goals

The focus of the audit was to verify that the smart contract system is secure, resilient, and working according to its specifications. The audit activities can be grouped into the following three categories:

1. Security: Identifying security-related issues within each contract and within the system of contracts.
2. Sound Architecture: Evaluation of the architecture of this system through the lens of established smart contract best practices and general software best practices.
3. Code Correctness and Quality: A full review of the contract source code. The primary areas of focus include
    a. Correctness
    b. Readability
    c. Sections of code with high complexity
    d. Quantity and quality of test coverage

Security Level Reference

Every issue in this report were assigned a severity level from the following:

High severity issues will bring problems and should be fixed.

Medium severity issues could potentially bring problems and should eventually be fixed.

Low severity issues are minor details and warnings that can remain unfixed but would be better fixed at some point in the future.

Issues	High	Medium	Low
Open	-	-	-
Closed	-	1	6

Contract: LinearPool.sol

High Severity Issues

No issues were found.

Medium Severity Issues

1. **Multiplication is being performed on the result of Division**
   Line No: 672-674

   Explanation:
   During the automated testing of the LinearPool contract, it was found that inside the **linearEarlyWithdraw()** function multiplication on the result of a division was performed.

   ```
   uint128 amount = (_percentage * stakingData.balance) / HUNDRED_PERCENT;
   uint128 penaltyFee = (amount * poolInfo.penaltyRate[_percentage]) /
    HUNDRED_PERCENT;
   ```

   Integer Divisions in Solidity might truncate and lead to a loss of precision.

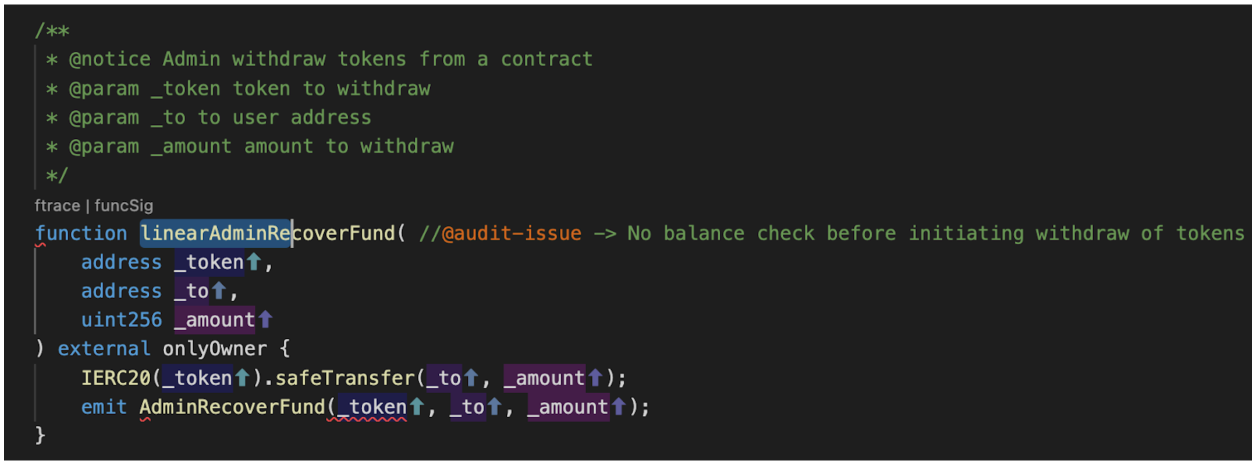
   Recommendation:
   Solidity doesn't encourage arithmetic operations that involve division before multiplication. Therefore the above-mentioned function should be tested for specific corner cases once and redesigned if they do not lead to the expected results.

   Amended (April 20th, 2022): The issue was fixed by the **Ekta** team and is no longer present in commit 951368e5ca2880fe67b52c860d3859907f0f0d48.

Low Severity Issues

1. **The contract's token balance is not checked before initiating a transfer**

   Explanation:
   The **linearAdminRecoverFund()** function in the contract doesn't verify whether or not the amount of token that is to be withdrawn by the owner is available in the contract, in the first place.



```
/**
 * @notice Admin withdraw tokens from a contract
 * @param _token token to withdraw
 * @param _to to user address
 * @param _amount amount to withdraw
 */
ftrace | funcSig
function linearAdminRecoverFund( //@audit-issue -> No balance check before initiating withdraw of tokens
    address _token,
    address _to,
    uint256 _amount
) external onlyOwner {
    IERC20(_token).safeTransfer(_to, _amount);
    emit AdminRecoverFund(_token, _to, _amount);
}
```

   While this doesn't affect the intended behavior of the contract, it will lead to a transaction revert without an adequate error message.

   Recommendation:
   It's considered better to include a **require** statement to validate that the contract has the **require** amount of tokens before triggering any token transfer from the contract.

   Amended (April 20th, 2022): The issue was fixed by the **Ekta** team and is no longer present in commit 951368e5ca2880fe67b52c860d3859907f0f0d48.

2. **Absence of Event Emission after imperative State variable Update**

   Explanation:
   Functions that update an imperative arithmetic state variable contract should emit an event after the update.

   As per the current architecture of the contract, the following functions update crucial state variables but don't emit any event on its modification:

- setTreasury()
- linearSetRewardDistributor()

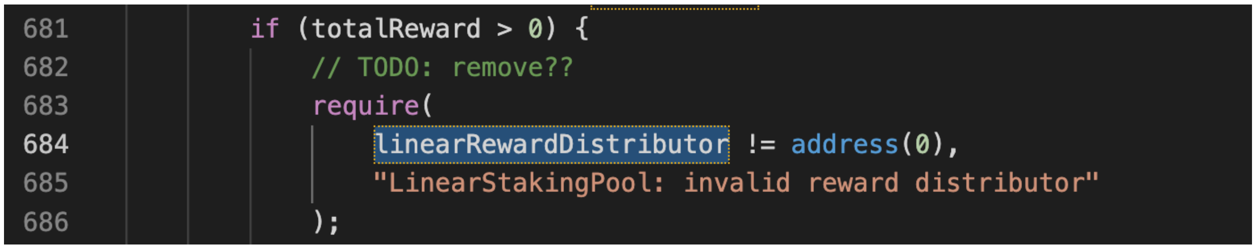
The absence of event emission for important state variables update also makes it difficult to track them off-chain as well.

Recommendation:
As per the best practices in smart contract development, an event should be fired after changing crucial arithmetic state variables.

Amended (April 20th, 2022): The issue was fixed by the **Ekta** team and is no longer present in commit 951368e5ca2880fe67b52c860d3859907f0f0d48.

3. Redundant require statement found in the linearEarlyWithdraw() function
Line no: 683-686

Explanation:
The **linearEarlyWithdraw()** function includes a require statement in the function body to check whether or not the linearRewardDistributor state variable is a valid address.



```
681         if (totalReward > 0) {
682             // TODO: remove??
683             require(
684                 linearRewardDistributor != address(0),
685                 "LinearStakingPool: invalid reward distributor"
686             );
```

However, this function doesn't invoke any transfer of tokens from linearRewardDistributor but only stores the rewards amount of the user in respective state variables. Therefore, this **require** statement doesn't have any significance in the function.

Recommendation:
Redundant use of **require** statements should be avoided in functions to optimize the code and enhance readability.

Amended (April 20th, 2022): The issue was fixed by the **Ekta** team and is no longer present in commit 951368e5ca2880fe67b52c860d3859907f0f0d48.

4. **Validity of linearRewardDistributor address is not checked in _linearClaimPendingWithdraw() function**
   Line no: 961-965

   Explanation:
   The **_linearClaimPendingWithdraw()** requires the reward amount to be transferred from the **linearRewardDistributor** address.

   However, unlike the pre-existing function designs of the LinearPool contract, the **_linearClaimPendingWithdraw()** function doesn't ensure whether or not the reward distributor address is a valid address and not a zero address.

   Recommendation:
   Before initiating any token transfer from the linearRewardDistributor address, a require statement must validate that the distributor address is not a zero address and has been set by the owner beforehand.

   Amended (April 20th, 2022): The issue was fixed by the **Ekta** team and is no longer present in commit 951368e5ca2880fe67b52c860d3859907f0f0d48.

5. **Violation of Check_Effects_Interaction Pattern in the linearEarlyWithdraw() function**
   Line no - 711

   Explanation:
   The **linearEarlyWithdraw() function** update important state variable of the contract after the external calls are being made.

   An external call within a function technically shifts the control flow of the contract to another contract for a particular period of time. Therefore, as per the Solidity Guidelines, any modification of the state variables in the base contract must be performed before executing the external call.

   Although the function has been assigned a **nonReentrant()** modifier, it is always a better practice to follow the best solidity practices while dealing with state variable modifications and external calls.

   Recommendation:
   Check Effects Interaction Pattern must be followed while implementing external calls in a function.

   Amended (April 20th, 2022): The issue was fixed by the **Ekta** team and is no longer present in commit 951368e5ca2880fe67b52c860d3859907f0f0d48.

6. **External Visibility should be preferred**

   Explanation:
   Functions that are never called throughout the contract should be marked as **external** visibility instead of **public** visibility.
   This will effectively result in Gas Optimization as well.

   Therefore, the following function must be marked as **external** within the contract:
   - **pauseContract()**
   - **linearClaimPendingWithdraw()**

   Recommendation:
   If the **PUBLIC** visibility of the above-mentioned functions is not intended, then the **EXTERNAL** Visibility keyword should be preferred.

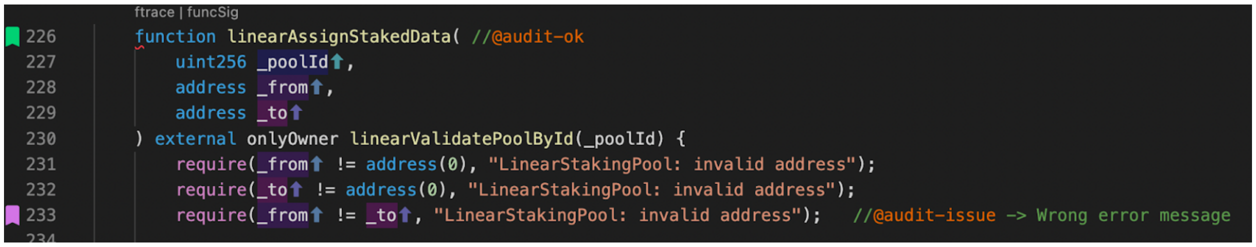
   Amended (April 20th, 2022): The issue was fixed by the **Ekta** team and is no longer present in commit 951368e5ca2880fe67b52c860d3859907f0f0d48.

Recommendations / Informational

1. **Inadequate Error messages found in require statements**
   Line no - 233

   Explanation:
   The **linearAssignStakedDate()** function in the contract includes a require statement to ensure that the _from and _to address are not exactly similar.

   However, the error message attached to this **require** statement indicates invalid information.



```
            ftrace | funcSig
226         function linearAssignStakedData( //@audit-ok
227             uint256 _poolId,
228             address _from,
229             address _to
230         ) external onlyOwner linearValidatePoolById(_poolId) {
231             require(_from != address(0), "LinearStakingPool: invalid address");
232             require(_to != address(0), "LinearStakingPool: invalid address");
233             require(_from != _to, "LinearStakingPool: invalid address");   //@audit-issue -> Wrong error message
```

   While this makes it troublesome to detect the reason behind a particular function revert, it also reduces the readability of the code.

   Recommendation:
   Adequate error Messages must be included in every **require** statement in the contract.

This audit does not provide a security or correctness guarantee of the audited smart contract. Securing smart contracts is a multistep process, therefore running a bug bounty program as a complement to this audit is strongly recommended.

Amended (April 20th, 2022): The issue was fixed by the **Ekta** team and is no longer present in commit 951368e5ca2880fe67b52c860d3859907f0f0d48.

2. **NatSpec Annotations must be included**
As the contact size exceeds 24576 so it cannot be deployed directly.

Recommendation:
Enabling the adequate compiler optimization runs, must be kept in mind for deploying properly.

Amended (April 20th, 2022): The issue was fixed by the **Ekta** team and is no longer present in commit 951368e5ca2880fe67b52c860d3859907f0f0d48.

3. **Coding Style Issues in the Contract**

Explanation:
Code readability of a Smart Contract is largely influenced by the Coding Style issues and in some specific scenarios may lead to bugs in the future.

During the automated testing, it was found that the **NAME** contract had quite a few code style issues.

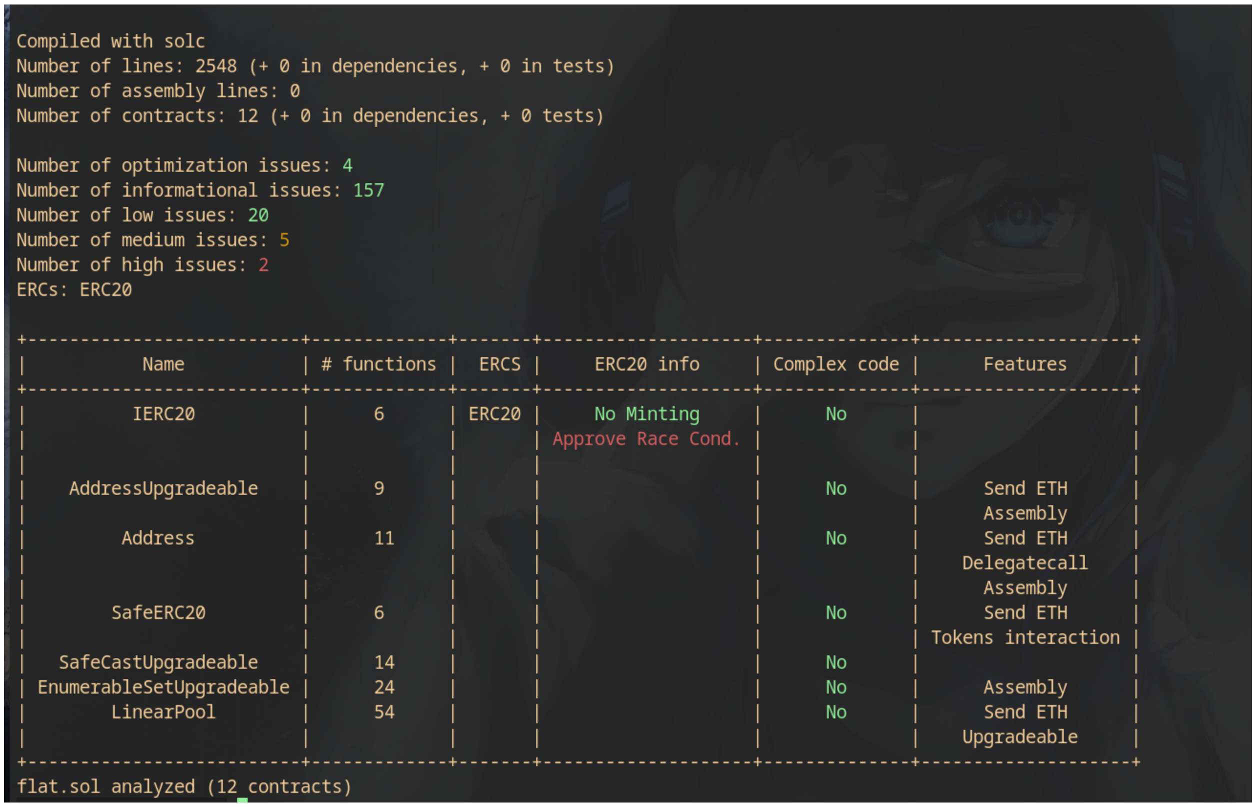
Recommendation:
Therefore, it is recommended to fix the issues like naming convention, indentation, and code layout issues in a smart contract.
From the solidity style guide, it is recommended to write the contract in the following order:

- Type declarations
- State variables
- Events
- Functions

More info here

Amended (April 20th, 2022): The issue was fixed by the **Ekta** team and is no longer present in commit 951368e5ca2880fe67b52c860d3859907f0f0d48.

Automated Audit Result

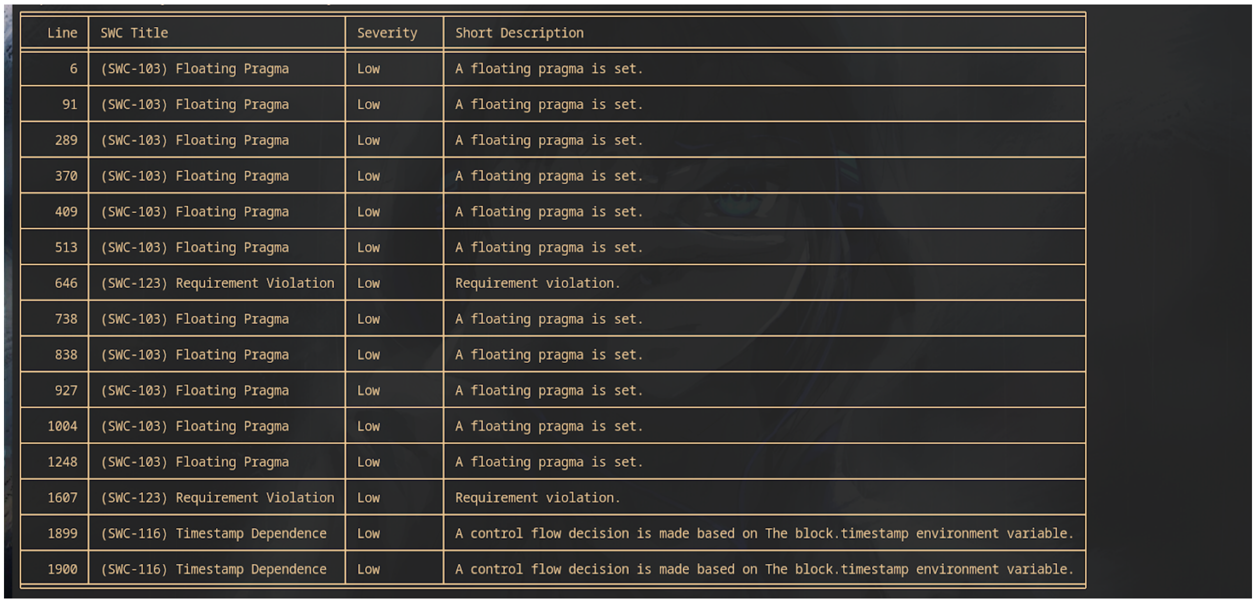


```
Compiled with solc
Number of lines: 2548 (+ 0 in dependencies, + 0 in tests)
Number of assembly lines: 0
Number of contracts: 12 (+ 0 in dependencies, + 0 tests)

Number of optimization issues: 4
Number of informational issues: 157
Number of low issues: 20
Number of medium issues: 5
Number of high issues: 2
ERCs: ERC20

+-------------------------+-------------+-------+--------------------+--------------+--------------------+
|          Name           | # functions | ERCS  |     ERC20 info     | Complex code |      Features      |
+-------------------------+-------------+-------+--------------------+--------------+--------------------+
|         IERC20          |      6      | ERC20 |     No Minting     |      No      |                    |
|                         |             |       | Approve Race Cond. |              |                    |
|                         |             |       |                    |              |                    |
|   AddressUpgradeable    |      9      |       |                    |      No      |      Send ETH      |
|                         |             |       |                    |              |      Assembly      |
|         Address         |     11      |       |                    |      No      |      Send ETH      |
|                         |             |       |                    |              |    Delegatecall    |
|                         |             |       |                    |              |      Assembly      |
|        SafeERC20        |      6      |       |                    |      No      |      Send ETH      |
|                         |             |       |                    |              | Tokens interaction |
|   SafeCastUpgradeable   |     14      |       |                    |      No      |                    |
| EnumerableSetUpgradeable|     24      |       |                    |      No      |      Assembly      |
|       LinearPool        |     54      |       |                    |      No      |      Send ETH      |
|                         |             |       |                    |              |    Upgradeable     |
+-------------------------+-------------+-------+--------------------+--------------+--------------------+
flat.sol analyzed (12 contracts)
```

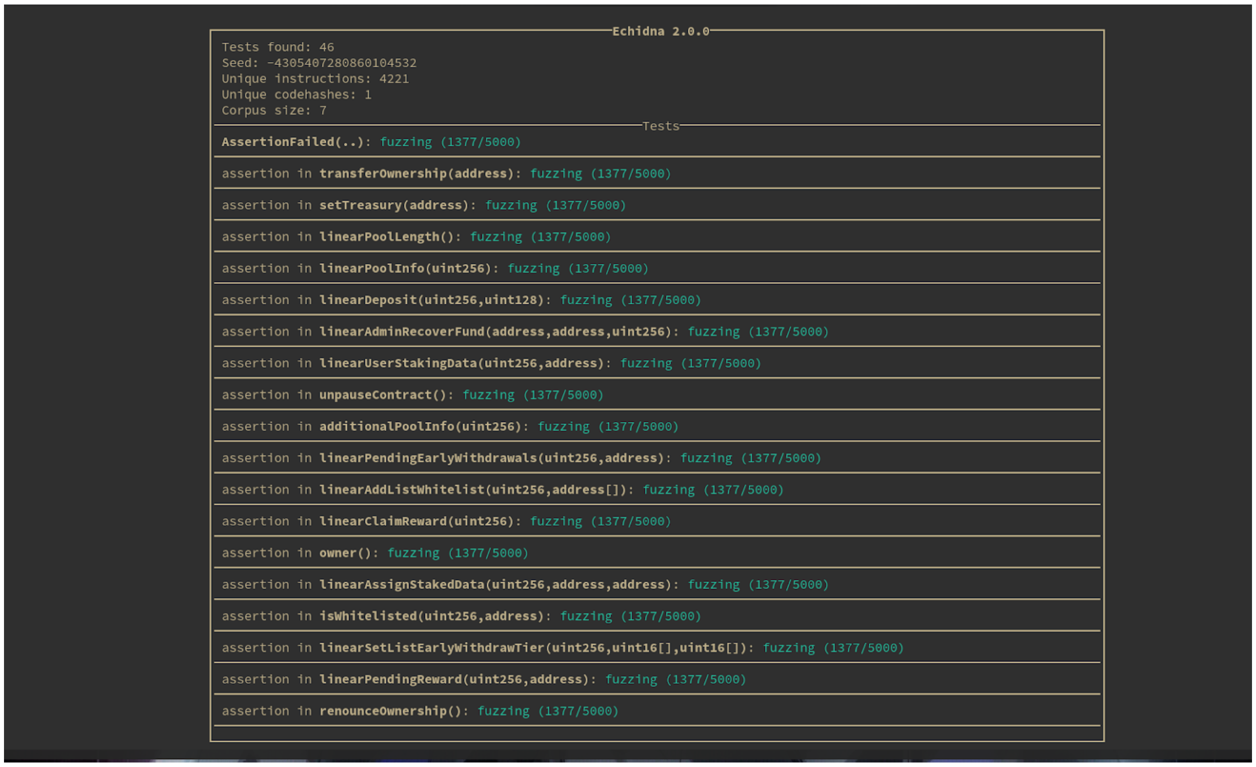
Mythx



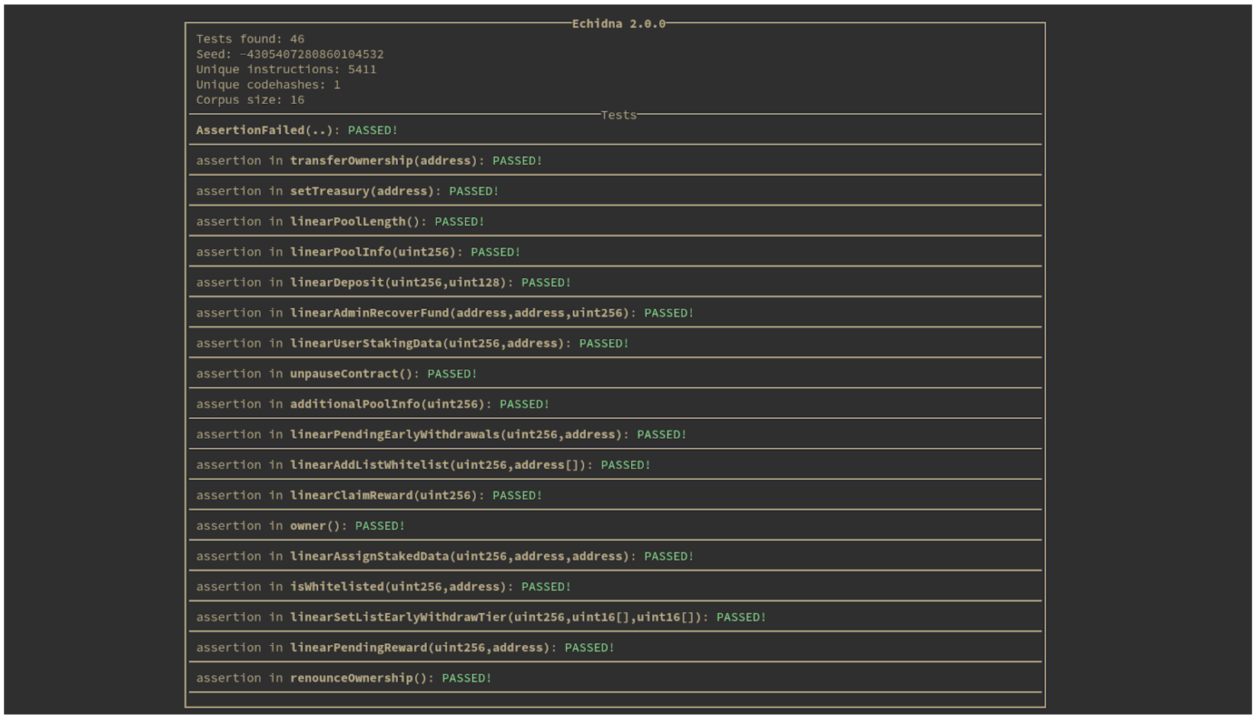
Line	SWC Title	Severity	Short Description
6	(SWC-103) Floating Pragma	Low	A floating pragma is set.
91	(SWC-103) Floating Pragma	Low	A floating pragma is set.
289	(SWC-103) Floating Pragma	Low	A floating pragma is set.
370	(SWC-103) Floating Pragma	Low	A floating pragma is set.
409	(SWC-103) Floating Pragma	Low	A floating pragma is set.
513	(SWC-103) Floating Pragma	Low	A floating pragma is set.
646	(SWC-123) Requirement Violation	Low	Requirement violation.
738	(SWC-103) Floating Pragma	Low	A floating pragma is set.
838	(SWC-103) Floating Pragma	Low	A floating pragma is set.
927	(SWC-103) Floating Pragma	Low	A floating pragma is set.
1004	(SWC-103) Floating Pragma	Low	A floating pragma is set.
1248	(SWC-103) Floating Pragma	Low	A floating pragma is set.
1607	(SWC-123) Requirement Violation	Low	Requirement violation.
1899	(SWC-116) Timestamp Dependence	Low	A control flow decision is made based on The block.timestamp environment variable.
1900	(SWC-116) Timestamp Dependence	Low	A control flow decision is made based on The block.timestamp environment variable.

This audit does not provide a security or correctness guarantee of the audited smart contract. Securing smart contracts is a multistep process, therefore running a bug bounty program as a complement to this audit is strongly recommended.

Echidna Assertion Test



```
Echidna 2.0.0
Tests found: 46
Seed: -4305407280860104532
Unique instructions: 4221
Unique codehashes: 1
Corpus size: 7
Tests
AssertionFailed(..): fuzzing (1377/5000)
assertion in transferOwnership(address): fuzzing (1377/5000)
assertion in setTreasury(address): fuzzing (1377/5000)
assertion in linearPoolLength(): fuzzing (1377/5000)
assertion in linearPoolInfo(uint256): fuzzing (1377/5000)
assertion in linearDeposit(uint256,uint128): fuzzing (1377/5000)
assertion in linearAdminRecoverFund(address,address,uint256): fuzzing (1377/5000)
assertion in linearUserStakingData(uint256,address): fuzzing (1377/5000)
assertion in unpauseContract(): fuzzing (1377/5000)
assertion in additionalPoolInfo(uint256): fuzzing (1377/5000)
assertion in linearPendingEarlyWithdrawals(uint256,address): fuzzing (1377/5000)
assertion in linearAddListWhitelist(uint256,address[]): fuzzing (1377/5000)
assertion in linearClaimReward(uint256): fuzzing (1377/5000)
assertion in owner(): fuzzing (1377/5000)
assertion in linearAssignStakedData(uint256,address,address): fuzzing (1377/5000)
assertion in isWhitelisted(uint256,address): fuzzing (1377/5000)
assertion in linearSetListEarlyWithdrawTier(uint256,uint16[],uint16[]): fuzzing (1377/5000)
assertion in linearPendingReward(uint256,address): fuzzing (1377/5000)
assertion in renounceOwnership(): fuzzing (1377/5000)
```



```
Echidna 2.0.0
Tests found: 46
Seed: -4305407280860104532
Unique instructions: 5411
Unique codehashes: 1
Corpus size: 16
Tests
AssertionFailed(..): PASSED!
assertion in transferOwnership(address): PASSED!
assertion in setTreasury(address): PASSED!
assertion in linearPoolLength(): PASSED!
assertion in linearPoolInfo(uint256): PASSED!
assertion in linearDeposit(uint256,uint128): PASSED!
assertion in linearAdminRecoverFund(address,address,uint256): PASSED!
assertion in linearUserStakingData(uint256,address): PASSED!
assertion in unpauseContract(): PASSED!
assertion in additionalPoolInfo(uint256): PASSED!
assertion in linearPendingEarlyWithdrawals(uint256,address): PASSED!
assertion in linearAddListWhitelist(uint256,address[]): PASSED!
assertion in linearClaimReward(uint256): PASSED!
assertion in owner(): PASSED!
assertion in linearAssignStakedData(uint256,address,address): PASSED!
assertion in isWhitelisted(uint256,address): PASSED!
assertion in linearSetListEarlyWithdrawTier(uint256,uint16[],uint16[]): PASSED!
assertion in linearPendingReward(uint256,address): PASSED!
assertion in renounceOwnership(): PASSED!
```

This audit does not provide a security or correctness guarantee of the audited smart contract. Securing smart contracts is a multistep process, therefore running a bug bounty program as a complement to this audit is strongly recommended.

Concluding Remarks

While conducting the audits of the Ekta smart contract, it was observed that the contracts contain Medium and Low severity issues.

Our auditors suggest that Medium and Low severity issues should be resolved by the developers. The recommendations given will improve the operations of the smart contract.

Disclaimer

ImmuneBytes's audit does not provide a security or correctness guarantee of the audited smart contract. Securing smart contracts is a multistep process, therefore running a bug bounty program as a complement to this audit is strongly recommended.

Our team does not endorse the Ekta platform or its product nor this audit is investment advice.
Notes:

- Please make sure contracts deployed on the mainnet are the ones audited.
- Check for the code refactor by the team on critical issues.

ImmuneBytes