Ekta

LinearPool, Mintable & AirDrop

Smart Contract Audit Report

December 15, 2021

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: AirDrop.sol, Mintable.sol
- Languages: Solidity(Smart contract)
- Github commit/Smart Contract Address for audit: Null
- 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 References

Every issue in this report was 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	1	3	9
Closed	-	-	-

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.

Contract: LinearPool.sol

High Severity Issues

1. **Inadequate implementation of delayDuration feature found in the LinearPool contract.**

Explanation:
The **LinearPool** contract includes a specific parameter named **delayDuration** while adding a new pool in the protocol.
This parameter is supposed to ensure that the users of a particular pool where the **delay duration** is greater than zero, must wait for a specific amount of time, as per the duration value, before receiving their payments while withdrawing their staked tokens from that pool.

However, during the manual code review of the contract, it was found that the **withdraw()** functions in the contract doesn't implement any effective logic to impose this delay of payment.

```
361         if (pool.delayDuration == 0) {
362             linearAcceptedToken.safeTransfer(account, _amount);
363             emit LinearWithdraw(_poolId, account, _amount);
364             return;
365         }
366
367         linearAcceptedToken.safeTransfer(account, _amount);
368     }
369
```

For instance, as can be seen in the code snippet above, the if statement at line 361 is executed if the **delayDuration** is exactly Zero. In this case, since there is no delay in payment, the tokens are transferred to the user right away, as expected.

However, if the **delayDuration** is greater than zero, there should have been some delay in the payments to the user, as per the intended behavior of the contract. However, no such implementation was found in any of the withdraw() functions. This might lead to an unwanted scenario where pools with a significant delay duration won't have any impact on the token transfer during the withdrawal procedure.

Recommendation:
If the LinearPool contract is not supposed to use the **delayDuration** parameter significantly, then it shall be removed from the contract. Otherwise, the delay feature should be implemented in the withdrawal functions of the contract.

Medium Severity Issues

1. **Improper require statement found in withdraw(uint256) function**
   Line no: 395-398

   Explanation:
   The linearWithdraw(uint256) function in the LinearPool Contract at line 374, allows a specific user to withdraw all his Staked Balance from the contract after the given **lockPeriod** is over.

```
394
395         require(
396             stakingData.balance >= stakingData.balance,
397             "LinearStakingPool: invalid withdraw amount"
398         );
399
```

   However, the function includes an inadequate require statement at the above-mentioned line number which indicates that the function shall only be called when the staked balance is either greater than or equal to itself.

   This doesn't represent a strong checkpoint as it leads to a scenario where users with ZERO balance staked are also being allowed to call this function, which should not be an intended scenario.

   Recommendation:
   An effective way to implement this require statement is to ensure that the staked balance is greater than ZERO.
   This ensures that the caller of the function has some staked balance and should be able to withdraw that entire staked balance through this function.

Low Severity Issues

1. **withdraw() functions in the contract doesn't ensure appropriate emission of LinearWithdraw() event**
   Line no: 363, 422

   Explanation:
   As per the current architecture of the contract, the LinearWithdraw() event is supposed to be emitted whenever a user is able to successfully withdraw their staked balance from the contract.

   However, the design of the withdraw functions in the contract (Line 316, 374) indicates that the **LinearWithdraw()** event is only emitted if the delay duration of a specific pool is strictly equal to ZERO. Additionally, if a specific pool has a **delayDuration** greater than zero, then the withdrawable amount of token is simply transferred to the user without any event emission.

```
420         if (pool.delayDuration == 0) {
421             linearAcceptedToken.safeTransfer(account, withdrawBalance);
422             emit LinearWithdraw(_poolId, account, withdrawBalance);
423             return;
424         }
425
426         linearAcceptedToken.safeTransfer(account, withdrawBalance);
427     }
```

   While this breaks the standard practice of emitting out imperative events on crucial state changes in the contract, it also makes it difficult to track correct details off-chain.

   Recommendation:
   It's highly recommended to ensure proper emission of events at crucial state-changing instances in the contract.

2. **Absence of input validations while adding a new pool**
   Line no: 165-207

   Explanation:
   The **linearAddPool()** doesn't include adequate input validations for some crucial arguments being passed to the function.

   While the functions impose checks on the _endJoinTime or _delayDuration arguments, it doesn't validate the **_maxInvestment, _maxInvestment, lockDuration** which are equally critical parameters for a pool.

Recommendation:
Adequate require statements for the above-mentioned parameters will not just filter out only valid arguments but also ensure that the uint values being passed are within a pre-defined range.

3. **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:

- **unpauseContract()**

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

Recommendations / Informational

1. **Contract includes functions with similar names**
Line no: 316, 374

Explanation:
During the audit procedure, it was found that the LinearPool includes functions with exactly similar names.
For instances, the contract includes withdraw functions with exactly similar names, i.e., **linearWithdraw()**.

While this affects the readability of the code, it isn't considered a better practice to name two or more functions with an exactly similar name.

Recommendation:

- Avoid using similar names for different functions.
- The **linearWithdraw(uint256 _poolId)** function allows a user to withdraw the total staked balance. Therefore, it could be renamed as **linearWithdrawAll()** or **linearTotalWithdraw()**.

Contract: Mintable.sol

High Severity Issues

No issues were found.

Medium Severity Issues

1. **Total Supply of the token will never reach its full capacity**
   Line no: 61-64

   Explanation:
   The **mint()** function of the contract includes a **require** statement to check that the number of tokens being minted never exceeds the **cap** limit, that is set during the contract initialization.

   However, as per the current design of the function, the **require** statement ensures that the **total supply** of the token, after minting, should be strictly less than the **cap limit**.

   For instance, if the maximum capacity set for the token (**cap value**) is **10000**, the total supply shall never exceed **9999**.

```
60
61      function mint(address user, uint256 amount) public whenNotPaused onlyMinter {
62          require(totalSupply() + amount < cap, "EKTA: FULL CAPACITY");
63          _mint(user, amount);
64      }
```

   This is due to the fact that the condition in the require statement includes a strict comparison sign (**<**) instead of (**<=**).

   Therefore, it leads to an unwanted scenario where the total supply of the token will never really reach its true capacity but will always be lesser than the **cap** value in the contract.

   Recommendation:
   If the above-mentioned scenario is not intentional, the condition in the **require** statement of the **mint()** function should include a greater than or equal to sign (**<=**) instead of a strictly greater than sign (**<**).

Low Severity Issues

1. **Redundant comparison to boolean constants**
   Line no: 106, 118

   Explanation:
   Boolean constants can directly be used in conditional statements or require statements.

```
101     function _beforeTokenTransfer(
102         address from,
103         address to,
104         uint256 amount
105     ) internal whenNotPaused override {
106         require(blacklist[from] == false, "BLACKLIST CANNOT TRANSFER");
107     }
108
```

   Therefore, it's not considered a better practice to explicitly use **TRUE or FALSE** in the **require** statements.

   Recommendation:
   The equality to boolean constants could be removed from the above-mentioned lines.

2. **Absence of Zero Address Validation**
   Line no- 74-77 & 79-82

   Explanation:
   The **MintableToken** Contract includes a few functions that update some of the imperative addresses in the contract like **minter**, **owner**, etc.

   However, during the code review of the contact it was found that no Zero Address Validation is implemented on the following functions while updating the address state variables of the contract:
   - **changeMinter()**
   - **changeOwner()**

   Recommendation:
   A **require** statement should be included in such functions to ensure no zero address is passed in the arguments.

3. **No Events emitted after imperative State Variable modification**
   Line no - 56-59

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

   The **setCapacity()** function in the contract modifies the **cap** value, which is a crucial arithmetic state variable in the protocol.

   However, the function doesn't emit any event for this variable update.
   Since there is no event emitted on updating these variables, it might be difficult to track it off-chain.

   Recommendation:
   An event should be fired after changing crucial arithmetic state variables.

4. **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:
   - **mint()**
   - **burn()**
   - **pauseContract()**
   - **unpauseContract()**
   - **changeMinter()**
   - **changeOwner()**
   - **addBlacklist()**
   - **removeBlacklist()**
   - **getOwner()**
   - **getMinter()**
   - **isBlacklisted()**

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

Recommendations / Informational

1. **Constructor keyword could be removed in an Upgradeable Contract**

   Explanation:
   As per the standard practice, while writing upgradeable contracts, it's imperative to use a one-time setter function, i.e., **initialize** function instead of the constructor.

   While the contract includes and uses an **initialize function** to set the imperative state variables, it also includes a constructor keyword in the contract which doesn't seem to have any significant usage.

   Recommendation:
   If the use of the constructor is not intentional, it can be removed from the contract.

Contract: AirDrop.sol

High Severity Issues

No issues were found.

Medium Severity Issues

1. **Inadequate ETH transfer function found in the contract**
   Line no: 175-180

   Explanation:
   The Airdrop contract includes a **_transfer()** function which initiates a transfer of to the given recipient address.

```
175     function _transfer(address _to, uint256 _amount) private {
176         address payable payableAddress = payable(address(uint160(_to)));
177         // solhint-disable-next-line
178         (bool success, ) = payableAddress.call{ value: _amount }("");
179         require(success, "POOL::TRANSFER_FEE_FAILED");
180     }
181
```

This transfer of ether is supposed to happen from the contract to the given recipient address. However, the Airdrop contract includes **fallback** functions with **revert()** statements which technically indicates that the contract is not supposed to receive any ETHER.

```
    // solhint-disable-next-line
    fallback() external {
        // solhint-disable-next-line
        revert();
    }

    receive() external payable {
        // solhint-disable-next-line
        revert();
    }
```

Moreover, during the review, it was found that no functions in the contract are allowed to receive ETHER. This indicates that the contract will never have enough ETHER to initiate an ether transfer to any address.

Recommendation:
The **_transfer** function should either be modified adequately or the contract should be allowed to receive ether in order to initiate eth transfers.

Low Severity Issues

1. **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:
   - **claimTokens()**
   - **claimAndStakeTokens()**

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

2. **Unused private functions**
   Line no: 175-180

   Explanation:
   During the review of the Airdrop contract, it was found that the **_transfer** function has been assigned **private** visibility which makes the function inaccessible from outside the contract.

   However, the **_transfer** function is never used throughout the contract as well.

   Recommendation:
   If the function has no significant usage in the contract, it must be removed.

Recommendations / Informational

--

Automated Audit Result

1. Airdrop Contract

```
Compiled with solc
Number of lines: 1118 (+ 0 in dependencies, + 0 in tests)
Number of assembly lines: 0
Number of contracts: 12 (+ 0 in dependencies, + 0 tests)

Number of optimization issues: 9
Number of informational issues: 62
Number of low issues: 4
Number of medium issues: 1
Number of high issues: 2
ERCs: ERC20

+----------------+-------------+-------+--------------------+--------------+--------------------+
|      Name      | # functions |  ERCS |     ERC20 info     | Complex code |      Features      |
+----------------+-------------+-------+--------------------+--------------+--------------------+
|     IERC20     |      6      | ERC20 |     No Minting     |      No      |                    |
|                |             |       | Approve Race Cond. |              |                    |
|                |             |       |                    |              |                    |
|  ILinearPool   |      5      |       |                    |      No      |                    |
| TransferHelper |      3      |       |                    |      No      |                    |
|    SafeMath    |      10     |       |                    |      No      |                    |
|     ECDSA      |      9      |       |                    |      No      |     Ecrecover      |
|                |             |       |                    |              |      Assembly      |
|    Airdrop     |      35     |       |                    |      No      |    Receive ETH     |
|                |             |       |                    |              |      Send ETH      |
|                |             |       |                    |              |     Ecrecover      |
|                |             |       |                    |              | Tokens interaction |
|                |             |       |                    |              |      Assembly      |
|                |             |       |                    |              |    Upgradeable     |
+----------------+-------------+-------+--------------------+--------------+--------------------+
INFO:Slither:airDropFlat.sol analyzed (12 contracts)
```

2. MintableToken Contract

```
Compiled with solc
Number of lines: 740 (+ 0 in dependencies, + 0 in tests)
Number of assembly lines: 0
Number of contracts: 7 (+ 0 in dependencies, + 0 tests)

Number of optimization issues: 22
Number of informational issues: 23
Number of low issues: 5
Number of medium issues: 0
Number of high issues: 2
ERCs: ERC20

+---------------+-------------+-------+--------------------+--------------+-------------+
|      Name     | # functions |  ERCS |     ERC20 info     | Complex code |   Features  |
+---------------+-------------+-------+--------------------+--------------+-------------+
| MintableToken |      53     | ERC20 |     ∞ Minting      |      No      | Upgradeable |
|               |             |       | Approve Race Cond. |              |             |
|               |             |       |                    |              |             |
+---------------+-------------+-------+--------------------+--------------+-------------+
INFO:Slither:mintableFlat.sol analyzed (7 contracts)
```

3. LinearPool Contract

```
Compiled with solc
Number of lines: 1919 (+ 0 in dependencies, + 0 in tests)
Number of assembly lines: 0
Number of contracts: 11 (+ 0 in dependencies, + 0 tests)

Number of optimization issues: 4
Number of informational issues: 110
Number of low issues: 16
Number of medium issues: 2
Number of high issues: 2
ERCs: ERC20

+--------------------------+-------------+-------+--------------------+--------------+--------------------+
|          Name            | # functions |  ERCS |      ERC20 info    | Complex code |      Features      |
+--------------------------+-------------+-------+--------------------+--------------+--------------------+
|         IERC20           |      6      | ERC20 |     No Minting     |      No      |                    |
|                          |             |       | Approve Race Cond. |              |                    |
|                          |             |       |                    |              |                    |
|         Address          |     11      |       |                    |      No      |      Send ETH      |
|                          |             |       |                    |              |    Delegatecall    |
|                          |             |       |                    |              |      Assembly      |
|        SafeERC20         |      6      |       |                    |      No      |      Send ETH      |
|                          |             |       |                    |              | Tokens interaction |
|   SafeCastUpgradeable    |     14      |       |                    |      No      |                    |
| EnumerableSetUpgradeable |     24      |       |                    |      No      |      Assembly      |
|        LinearPool        |     39      |       |                    |      No      |      Send ETH      |
|                          |             |       |                    |              |    Upgradeable     |
+--------------------------+-------------+-------+--------------------+--------------+--------------------+
INFO:Slither:linearPoolFlat.sol analyzed (11 contracts)
```

Concluding Remarks

While conducting the audits of the Ekta smart contracts, 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