BLOCKAPEX

SMART CONTRACT SECURITY

V 1.0

DATE: 19th APR 2024

PREPARED FOR: ELECTROSWAP

SECURITY REPORT

BLOCKAPEX VERIFIED

About BlockApex

Founded in early 2021, is a security-first blockchain consulting firm. We offer services in a wide range of areas including Audits for Smart Contracts, Blockchain Protocols, Tokenomics along with Invariant development (i.e., test-suite) and Decentralized Application Penetration Testing. With a dedicated team of over 40+ experts dispersed globally, BlockApex has contributed to enhancing the security of essential software components utilized by many users worldwide, including vital systems and technologies.

BlockApex has a focus on blockchain security, maintaining an expertise hub to navigate this dynamic field. We actively contribute to security research and openly share our findings with the community. Our work is available for review at our public repository, showcasing audit reports and insights into our innovative practices.

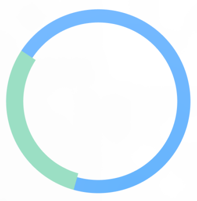
To stay informed about BlockApex's latest developments, breakthroughs, and services, we invite you to follow us on Twitter and explore our GitHub. For direct inquiries, partnership opportunities, or to learn more about how BlockApex can assist your organization in achieving its security objectives, please visit our Contact page at our website , or reach out to us via email at hello@blockapex.io.

Contents

1 Executive Summary

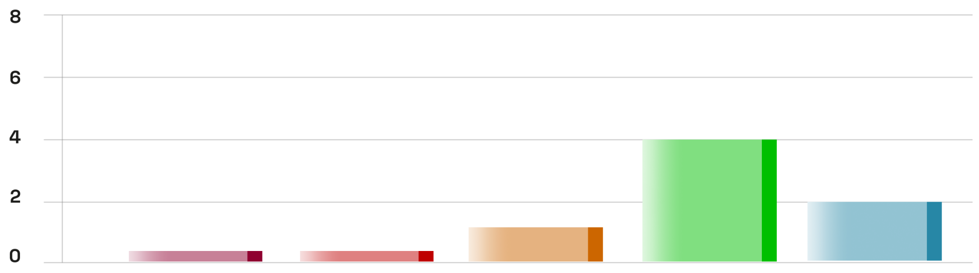
The audit was conducted on the ElectroSwapLockerV2 protocol, undertaken by two auditors auditor. This comprehensive review employed a meticulous manual code analysis approach, focusing exclusively on a line-by-line examination of the contract's code to identify potential vulnerabilities, coding best practices deviations, and areas for optimization.

Developer Response



●	Fixed	5
●	Acknowledged	0
●	Closed	2

Issues Overview



• **Critical**
Critical vulnerabilities are too easy to exploit and can lead to damages/loss in assets or manipulations.

• **High**
High-level vulnerabilities are difficult to exploit; however, they also have a significant impact on smart contract execution.

• **Medium**
Medium-level vulnerabilities are equally imperative to fix but they tend to have minimal impact on asset loss or data manipulations.

• **Low**
Lowest-level vulnerabilities, informational errors, violating code styles/ practices usually can't affect smart contract execution; hence they can be ignored.

• **Info**
Concerns that have no direct negative impact. These may help improve coding practices.

1.1 Scope

1.1.1 In Scope

The audit focuses on the ElectroSwapLockerV2 smart contract, which is designed for managing liquidity locks within the ElectroSwap protocol. This contract facilitates various functionalities related to liquidity locks such as creating, transferring, extending, and withdrawing locked liquidity. Key features include:

- Handling of liquidity pairs via the IElectroSwapPair and IElectroSwapFactory interfaces to ensure only eligible ElectroSwap liquidity pairs are locked.
- Fee management where fees can be configured and are applicable on operations like creating and increasing locks, with options to pay in multiple tokens.
- Access controls that restrict certain administrative functions to the team or contract owner.

Additionally, the lock migration contract, named ElectroSwapLockerV2Migrator.sol, was taken in scope for a manual code review. This contract allows users to migrate their LPToken locks to a new version of the ElectroSwapLocker smart contract, e.g. as mentioned in the comments, "to be used in future if Uniswap V# type liquidity support is added".

Contracts in Scope:

All Files under the folder: Contracts/ElectroSwapLockerV2/*

Initial Commit Hash: 1a2fd21688492df599593754ac80f57cf6273f8b

Final Commit Hash: d16787061c9539783abd0e6e1c1db71ddb9f4abf

ElectroSwapLockerV2 Deployed Link: https://blockexplorer.electroneum.com/address/0x16ca736c8B181772009e598F37f137e9cD36AFAE/contracts#address-tabs

ElectroSwapLockerV2Migrator Deployed Link: https://blockexplorer.electroneum.com/address/0xd99727Bb49E45f817882E47f132c8f64dB43A0dC/contracts#address-tabs

1.1.2 Out of Scope

All features or functionalities not delineated within the "In Scope" section of this document shall be deemed outside the review of this audit. This exclusion particularly applies to the backend operations of the ElectroSwapLockerV2 contracts associated with the platform & any other external libraries

1.2 Methodology

The audit of the ElectroSwap smart contract was conducted over the course of 5 days, utilizing a straightforward, manual code review approach by one auditor. The methodology began with a reconnaissance phase to gain an initial understanding of the contract's structure and intended functionalities. Following this, the auditor engaged in a detailed, line-by-line examination of the code. This manual review process focused on identifying logical flaws, security vulnerabilities, and opportunities for optimization, while ensuring adherence to Solidity best practices, security design patterns, and coding standards for clarity and efficiency.

1.3 Centralization Risks Analysis

The ElectroSwapLockerV2 smart contract, while designed to provide liquidity locking services, contains several points of centralization which could pose risks under certain conditions. These centralization points primarily stem from the role and permissions given to the teamWallet and the control mechanisms in place for managing the contract's operational parameters.

1.3.1 Key Points of Centralization:

- **Team Wallet Control**: The contract contains multiple functions that are gated by the `onlyTeam` modifier, which restricts access exclusively to the address stored in `teamWallet`. This centralized control includes critical functionalities such as:
  - Setting fees for transactions (`_setFees`).
  - Updating the team wallet address (`_setTeamWallet`).
  - Adding or removing addresses from the fee whitelist (`_updateFeeWhitelist`).
  - Setting the migrator contract address (`_setMigrator`).
- **Migrator Functionality**: The `_setMigrator` function enables the team to define a migrator contract that can interact with the locked funds. This function, if misused or compromised, poses a significant risk as it could enable the redirection of all locked funds to an arbitrary address.

1.4 Project Goals

The engagement was scoped to provide a security assessment of the ElectroSwapLockerV2 smart contract. Specifically, we sought to answer the following non-exhaustive list of questions:

1. Is there any reentrancy vulnerability present in the contract?
2. Does the liquidity locking mechanism have any edge cases or vulnerabilities?
3. Is there any risk associated with fee handling, particularly related to incorrect fee calculations or improper fee transfers?
4. Is there any possibility of unauthorized lock transfers or manipulations of lock ownership?
5. How does the contract handle state inconsistencies, especially in relation to lock records after operations like splits, increases, and migrations?
6. Can a scenario be created leveraging block.timestamp for a time manipulation attack?

1.5 Status Descriptions

Acknowledged: The issue has been recognized and is under review. It indicates that the relevant team is aware of the problem and is actively considering the next steps or solutions.

Fixed: The issue has been addressed and resolved. Necessary actions or corrections have been implemented to eliminate the vulnerability or problem.

Closed: This status signifies that the issue has been thoroughly evaluated and acknowledged by the development team. While no immediate action is being taken.

1.6 Summary of Findings Identified

S.No	Severity	Findings	Status
#1	MEDIUM	Potential Fund Drain Due to Reentrancy in migrateLock() Function	FIXED
#2	LOW	Desynchronized State Leads to Locked	FIXED
#3	LOW	Outdated Transaction Method in lock() Function	FIXED
#4	LOW	Unrestricted Fee Settings in _setFees()	FIXED
#5	LOW	Risk of Denial of Service (DoS) via Unbounded Loop Operations	CLOSED
#6	INFO	Redundant Conditional Logic in getFees() Function	CLOSED
#7	INFO	Absence of Zero Address Validation	FIXED

2 Findings and Risk Analysis

2.1 Potential Fund Drain Due to Reentrancy in migrateLock() Function

Severity: Medium

Status: Fixed

Location :

`contracts/ElectroSwapLockerV2/ElectroSwapLockerV2.sol`

Description The `migrateLock()` function in the ElectroSwapLockerV2 contract contains a reentrancy vulnerability due to the external call to a migrator contract followed by a state modification. This function transfers LP tokens to a specified migrator contract and then calls its migrate function. If the migrator contract is malicious, it could include a fallback function that calls back into `migrateLock()` functions of ElectroSwapLockerV2, potentially leading to drainage of funds. We consider this as an medium issue because the migrator will be trusted entiry but reentrancy is still possible.

Impact it allows an attacker to potentially drain all LP tokens locked within the contract using a minimal initial amount of locked tokens.

Proof of Concept :

1. The attacker deploys a malicious migrator contract.
2. The attacker triggers `migrateLock()` with a legitimate lockID.
3. Inside the malicious migrator's migrate function, a call is made back to `migrateLock()` vulnerable function of ElectroSwapLockerV2.
4. This reentry could drain LP tokens, exploiting the state changes post the external call.
5. This is applicable to every lp pair exist in the contract.

```
function migrateLock(uint _lockID) external {
        require(address(migrator) != address(0), &quot;Migrator not set&quot;);

        LockInfo storage lockInfo = locksById[_lockID];
        require(lockInfo.owner == msg.sender, &quot;Caller is not the owner of the lock&
            quot;);

        // Transfer the LP tokens to the migrator contract
        TransferHelper.safeTransfer(lockInfo.pair, address(migrator), lockInfo.amount);

        // Attempt to migrate the tokens
        require(migrator.migrate(lockInfo.pair, lockInfo.owner, lockInfo.amount, lockInfo.
            created, lockInfo.duration), &quot;Migration failed&quot;);

        // Mark the lock as empty
        lockInfo.amount = 0;

        emit LockMigrated(_lockID);
    }
```

Recommendation :

1. Use the `nonReentrant` modifier from the ReentrancyGuard contract for the `migrateLock()` function to prevent reentrancy.
2. Ensure that all state changes occur before calling external contracts.

2.2 Outdated Transaction Method in lock() Function

Severity: Low

Status: Fixed

Location :

`contracts/ElectroSwapLockerV2/ElectroSwapLockerV2.sol`

Description The `lock()` function in the ElectroSwapLockerV2 contract uses the `.transfer` method to send native token . Post EIP-1884 , the use of `.transfer` is discouraged due to its fixed gas stipend of 2300 gas, which may not be sufficient for all transactions, particularly in contracts that perform state changes or emit events in their fallback functions. This limitation can lead to failed transactions if the receiving contracts require more than 2300 gas.

Proof of Concept :

```
teamWallet.transfer(msg.value);
payable(msg.sender).transfer(msg.value);
```

Recommendation :

Replace the `.transfer` method with `.call.value(msg.value)(&quot;&quot;)` to send Native token. This change provides the flexibility to send all available gas or limit the gas based on the transaction's needs.

2.3 Precision Loss in Liquidity Fee Calculation Allows Protocol Fee Bypass

Severity: Low

Status: Fixed

Location :

`contracts/ElectroSwapLockerV2/ElectroSwapLockerV2.sol`

Description In the `lock()` and `increaseLock()` functions, the calculation for liquidity fees can result in zero due to precision loss when certain token amounts are locked. Specifically, locking or increasing by an amount that results in a sub-minimal fee calculation (e.g., 99 tokens with a 1% fee rate) leads to a calculated fee of zero. This issue arises because Solidity handles division by truncating results to the nearest lower integer, effectively allowing users to bypass protocol fees.

Impact While the direct financial impact per transaction might be minimal, cumulatively or through systematic exploitation, this could lead to significant losses of fee revenue for the protocol. This fee bypass undermines the economic model of the contract and potentially benefits a few at the expense of the overall system integrity.

Proof of Concept

```
liquidityFee = ((_additionalTokens * 100) * fees.lpPercentFee) / 10000;
TransferHelper.safeTransfer(lockInfo.pair, teamWallet, liquidityFee);
```

Using an example where `_amountToLock` = 99 tokens and fees.lpPercentFee = 100 (representing 1%):

1. The calculated liquidity fee should be 0.99, but due to Solidity's integer division, the result is truncated to 0.
2. This means no fee is charged for such transactions, leading to a potential loss of expected revenue for the protocol.

Recommendation :

1. **Rounding Up Fees**: Modify the fee calculation to round up the result, ensuring that any non-zero calculated fee results in at least the minimum chargeable fee.
2. **Restructuring Fee Calculation**: Consider using a fee calculation that adjusts the divisor or multipliers to reduce precision loss for small amounts.

2.4 Unrestricted Fee Settings in _setFees()

Severity: Low

Status: Fixed

Location :

`contracts/ElectroSwapLockerV2/ElectroSwapLockerV2.sol`

Description The `_setFees()` function in the ElectroSwapLockerV2 contract lacks sufficient input validation, allowing the onlyTeam authorized users to set arbitrarily high fee values for `etnFlatFee`, `boltFlatFee`, and `lpPercentFee`. This oversight could potentially allow for the setting of exorbitantly high fees, directly impacting users by charging unreasonably high amounts for operations involving liquidity locks.

Proof of Concept

```
function _setFees(uint _etnFlatFee, uint _boltFlatFee, uint _lpPercentFee) external
    onlyTeam {
       fees.etnFlatFee = _etnFlatFee;
       fees.boltFlatFee = _boltFlatFee;
       fees.lpPercentFee = _lpPercentFee;
   }
```

Recommendation :

Define reasonable maximum values for each fee type (`etnFlatFee`, `boltFlatFee`, `lpPercentFee`) and enforce these caps within the `_setFees()` function:

2.5 Risk of Denial of Service (DoS) via Unbounded Loop Operations

Severity: Low

Status: Closed

Location :

`contracts/ElectroSwapLockerV2/ElectroSwapLockerV2.sol`

Description The `withdraw()` function in the ElectroSwapLockerV2 contract contains a potential Denial of Service (DoS) vulnerability due to unbounded loop operations when updating arrays `lockIDsByPairAddress` and `lockIDsByOwner`. These operations occur when removing a lock reference after a withdrawal, which can lead to excessive gas costs if the arrays become excessively large, thereby causing transactions to fail due to out-of-gas errors or requiring prohibitively high gas fees

Recommendation :

Optimize the loop operations in `_removeLockReferenceFromPair()` and `_removeLockReferenceFromOwner()` to reduce their gas cost, possibly by reordering elements or by other means of minimizing array manipulations.

Developer Response

We do not expect that we will ever have enough concurrent locks that this would ever become a concern. The flat rate fee should protect against spamming of locks being created.

2.6 Redundant Conditional Logic in getFees() Function

Severity: Info

Status: Closed

Location :

`contracts/ElectroSwapLockerV2/ElectroSwapLockerV2.sol`

Description The `getFees()` function in the ElectroSwapLockerV2 contract includes an unnecessary boolean parameter `(_noop)` that does not alter the function's behavior, as the function returns the same fees struct regardless of the boolean's value. This redundancy can lead to confusion , as users may expect different behaviors based on the parameter value.

Proof of Concept

```
 function getFees(bool _noop) external view returns (Fees memory){
        return _noop ? fees : fees;
    }
```

Recommendation :

Since the boolean parameter `_noop` does not influence the outcome, it should be removed from the function signature:

```
function getFees() external view returns (Fees memory) {
    return fees;
}
```

Developer Response

The 'noop' input was put in place as a work around for the Electroneum block explorer only displaying return value correctly given some type of input.

2.7 Absence of Zero Address Validation

Severity: Info

Status: Fixed

Location :

`contracts/ElectroSwapLockerV2/ElectroSwapLockerV2.sol`

Description The contract lacks necessary validations to prevent setting the zero address (0x0) in functions `_setTeamWallet` and `transferLock`. Without these checks, there is a risk that critical functionalities, such as wallet management and lock ownership transfer, could be unintentionally assigned to the zero address, leading to irreversible loss of control and potential fund lockup.

Proof of Concept

```
function _setTeamWallet(address payable _teamWallet) external onlyTeam {
    teamWallet = _teamWallet;
}

function transferLock(uint _lockID, address _newOwner) external nonReentrant {
    LockInfo storage lockInfo = locksById[_lockID];
    require(lockInfo.owner == msg.sender, &quot;Caller is not the current owner of the
        lock&quot;);
    lockInfo.owner = _newOwner;
    lockIDsByOwner[_newOwner].push(_lockID);
    emit LockTransferred(_lockID, msg.sender, _newOwner);
}
```

Recommendation :

Implement zero address checks in both `_setTeamWallet` and `transferLock` functions to prevent invalid operations: