BLOCKAPEX

SMART CONTRACT SECURITY

V 1.0

SECURITY REPORT

BLOCKAPEX VERIFIED

DATE: 1st JULY 2024

PREPARED FOR: ELECTROSWAP

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 4

1.1 Scope . . . 5

1.1.1 In Scope . . . 5

1.1.2 Out of Scope . . . 6

1.2 Methodology . . . 6

1.3 Project Goals . . . 7

1.4 Status Descriptions . . . 8

1.5 Summary of Findings Identified . . . 9

2 Findings and Risk Analysis 10

2.1 Oversight in Lock Splitting Allows Users to Withdraw and Take Early Exit from the System. 10

2.2 Inflexible Liquidity Migration Settings can lead to Financial Inefficiencies. . . . 11

2.3 EOA Authorization in migrate Function Enables Front-Running Attack. . . . 12

2.4 migrateLock( ) Function is Broken Due to Data Reset Leading to Migration Failures. . . 13

2.5 Risk of Denial of Service via Unbounded loop Operations. . . . 14

2.6 Limitations of Fixed TeamWallet Address in ElectroSwapLockerV3Migrator Contract. . 16

2.7 Lack of Event Emissions in Key Functions. . . . 17

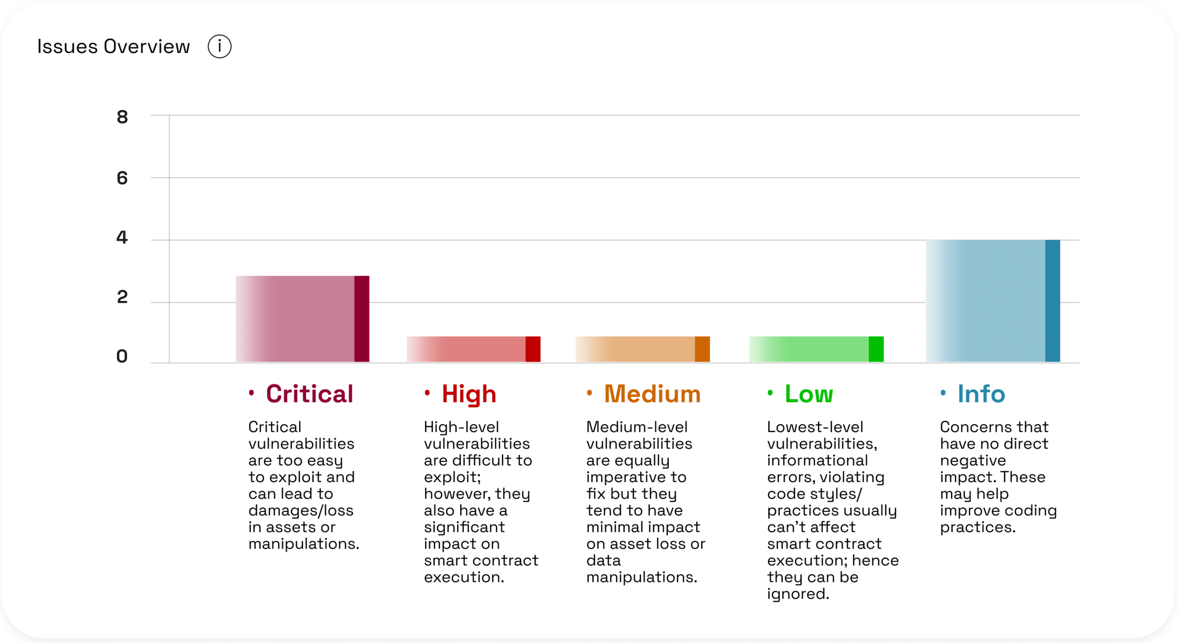
2.8 Fee variable should be transparent to users. . . . 18

2.9 Non-standard Function Naming Conventions . . . 19

2.10 Redundant Address Casting of Variables. . . . 20

1 Executive Summary

Our team performed a technique called Filtered Audit, where two individuals separately audited the ElectroSwap V3 Locker Contracts. After a thorough and rigorous manual testing process involving line by line code review for bugs, an automated tool-based review was carried out. All the raised flags were manually reviewed and re-tested to identify any false positives.



1.1 Scope

1.1.1 In Scope

The audit focuses on the ElectroSwapLockerV3 and ElectroSwapLockerV3Migrator smart contracts, which are designed for managing and migrating liquidity locks within the ElectroSwap protocol. These contracts facilitate various functionalities related to liquidity locks and the migration of liquidity from the V2 to the V3 protocol. Key features and functionalities of each contract are as follows:

ElectroSwapLockerV3 The ElectroSwapLockerV3 contract is responsible for managing liquidity locks within the ElectroSwap protocol. Its primary functionalities include:

- **Liquidity Lock Management:** Facilitates the creation, transfer, extension, and withdrawal of locked liquidity.
- **Handling V3 Liquidity Tokens:** Ensures that only eligible ElectroSwap liquidity pairs are locked.
- **Fee Management:** Configurable fees applicable to operations such as creating locks, with options to pay in native and Bolt tokens.
- **Access Controls:** Restricts certain administrative functions to the team or contract owner, ensuring secure management of critical operations.

ElectroSwapLockerV3Migrator The ElectroSwapLockerV3Migrator contract is responsible for migrating liquidity from the V2 protocol to the V3 protocol within the ElectroSwap ecosystem. Its primary functionalities include:

- **Liquidity Migration:** Manages the process of migrating liquidity from V2 to V3, ensuring a seamless transition.
- **Pool and Tick Management:** Initializes pools if they do not exist and handles the creation of new positions with specified fee tiers and tick ranges.
- **Access Controls:** Only authorized sources can initiate migrations, ensuring that the migration process is secure and controlled.
- **Configuration Management:** Allows for setting new pool defaults and managing source lockers, ensuring flexibility in managing liquidity migration operations.

Contracts in Scope: All Files under the folder: ElectroSwapLockerV3/*

Initial Commit Hash: 4319366bd0be41d3d0d6443da95bd2b989e6c1ca

Final Commit Hash: 0b70122af538266bdd10d58f7895866c3f94c317

Deployed Smart Contracts:

- **ElectroSwapLockerV3**
- **ElectroSwapLockerV3Migrator**

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 codebase was audited using a filtered audit technique. A band of two (2) auditors scanned the codebase in an iterative process for a time spanning 1 week. Starting with the recon phase, a basic understanding was developed, and the auditors worked on developing presumptions for the developed codebase and the relevant documentation/whitepaper. Furthermore, the audit moved on with the manual code reviews to find logical flaws in the codebase complemented with code optimizations, software, and security design patterns, code styles and best practices.

1.3 Project Goals

The engagement was scoped to provide a comprehensive security assessment of ElectroSwapLockerV3 contract. Specifically, we sought to answer the following non-exhaustive list of questions:

1. Are there any vulnerabilities in the ElectroSwapLockerV3 contract, such as reentrancy attacks, integer overflows/underflows, or improper input validation?
2. Does the contract ensure that only authorized entities can perform sensitive operations?
3. Are the roles and permissions correctly implemented and enforced?
4. Are the mechanisms for liquidity locking and splitting secure and resistant to potential exploits?
5. Is the fee management system implemented correctly, ensuring accurate calculations and preventing abuse?
6. Are event emissions adequate for monitoring state changes and ensuring transparency in critical operations?
7. Are there any gas optimization opportunities to improve the efficiency of the contract's functions?
8. Is the fixed fee tier and tick range design in the contract potentially leading to suboptimal user outcomes or financial losses?
9. Does the contract have appropriate mechanisms to update critical addresses like teamWallet to adapt to potential future needs?

1.4 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.

1.5 Summary of Findings Identified

S.No	Severity	Findings	Status
#1	CRITICAL	Oversight in Lock Splitting Allows Users to Withdraw and Take Early Exit from the System.	FIXED
#2	CRITICAL	Inflexible Liquidity Migration Settings can lead to Financial Inefficiencies.	FIXED
#3	CRITICAL	EOA Authorization in migrate Function Enables Front-Running Attack.	FIXED
#4	HIGH	migrateLock( ) Function is Broken Due to Data Reset Leading to Migration Failures.	FIXED
#5	MEDIUM	Risk of Denial of Service via Unbounded loop Operations.	CLOSED
#6	LOW	Limitations of Fixed TeamWallet Address in ElectroSwapLockerV3Migrator Contract.	FIXED
#7	INFO	Lack of Event Emissions in Key Functions.	FIXED
#8	INFO	Fee variable should be transparent to users.	FIXED
#9	INFO	Non-standard Function Naming Conventions.	FIXED
#10	INFO	Redundant Address Casting of Variables.	FIXED

2 Findings and Risk Analysis

2.1 Oversight in Lock Splitting Allows Users to Withdraw and Take Early Exit from the System.

Severity: Critical

Status: Fixed

Location ElectroSwapLockerV3/ElectroSwapLockerV3.sol

Description :

The **SplitLock** function allows users to split **100%** of the locked position into a new position with any new duration of time, which can be as short as one minute.

- **Attack Scenario:** A user locks a position with a certain duration, say for a month, intending to commit their liquidity for that period. After a very short period, even as soon as the next day, the user decides to split the entire position (100% of it) into a new position. During the split, the user sets a new lock duration for the new position, which can be as short as one minute. By transferring all the liquidity to a new position with a minimal lock duration, the user can then wait for this period to pass and regain full control over their liquidity, completely bypassing the originally intended lock duration.

Proof of Concept

```
function testSplitTheSplitedLockAndUnlock() public {
    positionManager.approve(address(lockerV3), tokenId);
    bolt.approve(address(lockerV3), type(uint256).max);
    lockerV3.lockPosition(tokenId, 10 days, false);
    lockerV3.splitLock(tokenId, 100, 1 days);
    vm.warp(block.timestamp + 1 days + 1 seconds);
    lockerV3.unlockPosition(tokenId + 1);
}
```

Recommendation :

1. **Restrict Liquidity Splitting Proportions:** Restrict the amount of liquidity that can be split off from a locked position or remove the previous lock if 100% amount is split into a new lock. For example, limit users to splitting off only a certain percentage of the total locked liquidity at any given time. If allowing 100% of the amount to be split into a new position, make sure to remove the old lock.
2. **Inherit Original Lock Duration:** Ensure that any new position created from a split inherits the lock duration of the original position, or at least does not allow a lock duration that ends before the original lock's expiration.

2.2 Inflexible Liquidity Migration Settings can lead to Financial Inefficiencies.

Severity: Critical

Status: Fixed

Location ElectroSwapLockerV3/ElectroSwapLockerV3Migrator.sol

Description :

Fixed Fee Tier: The **DEFAULT_FEE** is set to **3000**, representing a **0.3%** fee tier. In Uniswap V3, pools can have different fee tiers (e.g., **0.05%**, **0.3%**, **1.0%**). If a corresponding V3 pool with a **0.3%** fee does not exist, the migration function would be forced to create a new pool with **0.3%** fee tier. If the migration leads to the creation of a new pool due to the fixed fee setting, it might split the liquidity and trading volume across multiple pools, reducing potential fee earnings for all liquidity providers involved in those pools.

Fixed Price Range (Ticks): The **DEFAULT_TICK_LOWER** and **DEFAULT_TICK_UPPER** represent very broad price ranges which are currently defined in the contract. In Uniswap V3, liquidity providers can select specific price ranges to provide liquidity, allowing them to concentrate liquidity and potentially achieve higher returns. By using the range defined in migration contract, the contract may place liquidity on a whole graph, which could be potentially no returns for the liquidity provider.

Recommendation :

Dynamic Fee and Tick Configuration: Allow users or an automated system to determine the fee, tick-Lower, and tickUpper settings dynamically based on the current market conditions, user preferences, or historical data. This could be achieved by passing these parameters as arguments to the migration function.

```
function migrate(
    address pairAddress,
    address owner,
    uint256 amountToMigrate,
    uint256 lockCreated,
    uint256 lockDuration,
    uint24 fee,
    int24 tickLower,
    int24 tickUpper
) external onlyAuthorizedSource returns (bool) {
    // Migration logic using user-defined parameters
}
```

2.3 EOA Authorization in migrate Function Enables Front-Running Attack.

Severity: Critical

Status: Fixed

Location ElectroSwapLockerV3/ElectroSwapLockerV3Migrator.sol

Description :

The **migrate()** function in the **ElectroSwapLockerV3Migrator** contract performs critical operations to migrate liquidity, which includes several steps. Only an AuthorizedSource can migrate liquidity due to the access modifier. Consider if a whitelisted source could be an externally owned account (EOA) rather than a contract; it opens up potential front-running risks. The EOA or any authorized source needs to transfer the required LP tokens to this contract before invoking **migrate()**. A malicious actor could see a migrate transaction before it is confirmed and attempt to place their transaction before that. An attacker can potentially use the user's deposited LPs to create his position, and the user's transaction will fail due to the check **IERC20(pairAddress).balanceOf(address(this)) >= amountToMigrate**. This requires careful coordination between the token transfer and the migration transaction, which can be error-prone and potentially manipulated by other observers on the network.

Recommendation :

Limit the onlyAuthorizedSource whitelist to only include smart contracts rather than EOAs. This approach leverages contract code to enforce additional checks and validations, and logic that can reduce the risk.

2.4 migrateLock( ) Function is Broken Due to Data Reset Leading to Migration Failures.

Severity: High

Status: Fixed

Location ElectroSwapLockerV3/ElectroSwapLockerV3.sol

Description :

In the **migrateLock** function, the **lockInfo** variable is a reference to a storage slot corresponding to **lockedPositions[tokenId]**. This means **lockInfo** is directly pointing to the storage location of the lock information for the given **tokenId**. When the migrateLock function deletes the **lockedPositions[tokenId]**, it sets the entire storage slot to its default values, which for a struct would be the default values for each of its fields. This means that after this operation, all fields in **lockInfo** (such as **lockInfo.pool**, **lockInfo.owner**, **lockInfo.created**, and **lockInfo.duration**) would be reset to their respective default values (e.g., address(0) for addresses, 0 for uints). When **migrator.migrate(...)** is called, all the arguments passing are essentially zero or default values for all the parameters that are derived from **lockInfo**. This would likely result in the migration function not performing as intended.

Proof of Concept

```
function testMigratLock() public {
    lockerV3._setMigrator(address(lockerV3Migrator));
    positionManager.approve(address(lockerV3), tokenId);
    bolt.approve(address(lockerV3), type(uint256).max);
    lockerV3.lockPosition(tokenId, 10 days, false);
    vm.warp(block.timestamp + 11 days);
    vm.expectRevert();
    lockerV3.migrateLock(tokenId);
}
```

Results

```
ElectroSwapLockerV3Migrator::migrate(0x0000000000000000000000000000000000000000, 0
    x0000000000000000000000000000000000000000, 743699 [7.436e5], 0, 0)
    revert: Cannot migrate matured locks, withdraw instead
    ()
```

Recommendation :

Move the delete **lockedPositions[tokenId]** operation to after the migration has been successfully completed.

2.5 Risk of Denial of Service via Unbounded loop Operations.

Severity: Medium

Status: Closed

Location ElectroSwapLockerV3/ElectroSwapLockerV3.sol

Description :

The current implementation of **removeLockId()** utilizes arrays (**ownerToLocks** and **poolToLocks**) to manage lock IDs associated with owners and pools. This design requires iterating over potentially large arrays to find and remove specific lock IDs. When the array size becomes large, the iteration process can consume a significant amount of gas. In scenarios where the array is extremely large, the gas required for a single operation such as removing a lock ID might exceed the block gas limit. This can lead to transactions that consistently fail due to hitting these limits, effectively causing a Denial of Service (DoS).

Recommendation :

It is recommended to switch from using arrays to mappings for managing lock IDs. Mappings provide constant time complexity for adding, removing, and checking the existence of elements, which makes gas costs predictable and independent of the number of items managed.

Data Structure: Utilize a nested mapping structure where each address maps to another mapping, which then maps lock IDs to a boolean or a struct indicating the presence and status of the lock.

```
mapping(address =&gt; mapping(uint256 =&gt; bool)) private lockIsActive;
```

Adding Lock IDs: Simply set the value in the nested mapping to true when a lock is added

```
lockIsActive[_address][_lockId] = true;
```

Removing Lock IDs: Set the value to false when a lock is removed.

```
lockIsActive[_address][_lockId] = false;
```

Checking Existence: Check the status of a lock ID in constant time by referencing the mapping.

```
bool isActive = lockIsActive[_address][_lockId];
```

Implementing the mapping eliminates the risk of exceeding block gas limits, as there is no need for iteration over elements.

Developer Response: Suggested implementation breaks other requirements (getLockIdsByOwner, getLockIdsByPool) with no easy work around. Extremely unlikely to occur as in most cases there's only 1-2 locks per liquidity pair/pool. Expired locks reduce size of array, so it will only represent active locks. For these reasons and the very low chances of occuring, leaving as-is.

Auditor Response We understand the decision to leave as it is due to the low likelihood of occurrence.

2.6 Limitations of Fixed TeamWallet Address in ElectroSwapLockerV3Migrator Contract.

Severity: Low

Status: Fixed

Location ElectroSwapLockerV3/ElectroSwapLockerV3.sol

Description :

In the **ElectroSwapLockerV3Migrator** contract, the **teamWallet** address is set during the construction of the contract and is not modifiable afterward. This design lacks flexibility and does not account for scenarios where changing the wallet address may be necessary, such as:

- Security breaches where the private keys of the current wallet are compromised.
- Organizational changes that require transitioning financial operations to a new wallet.
- Upgrading to a more secure wallet setup, such as a multi-signature wallet for better control and security.

Recommendation :

Implement a function that allows changing the **teamWallet**. Ensure that this function can only be called by the current **teamWallet** or through a governance mechanism that involves multiple stakeholders to prevent unauthorized access.

```
event TeamWalletUpdated(address indexed oldWallet, address indexed newWallet);

function setTeamWallet(address payable newWallet) external onlyTeam {
    require(newWallet != address(0), &quot;Invalid address&quot;);
    emit TeamWalletUpdated(teamWallet, newWallet);
    teamWallet = newWallet;
}
```

2.7 Lack of Event Emissions in Key Functions.

Severity: Info

Status: Fixed

Description :

Several functions in the ElectroSwapLockerV3 and ElectroSwapLockerV3Migrator contracts perform significant state-changing actions but do not emit events. For reference.

ElectroSwapLockerV3: _setFees() _setTeamWallet() _updateFeeWhitelist() _setMigrator()

ElectroSwapLockerV3Migrator: _setLocker() _setSourceLocker() migrate()

Recommendation :

Add events to these functions to log important state changes for better transparency and tracking.

2.8 Fee variable should be transparent to users.

Severity: Info

Status: Fixed

Location ElectroSwapLockerV3/ElectroSwapLockerV3/sol

Description :

The fees variable in the ElectroSwapLockerV3 contract is set to private, making the fee structure inaccessible to anyone outside the contract. This lack of accessibility prevents users from viewing or verifying the applied fees.

Code Affected

```
Fees private fees;
```

Recommendation :

Change the visibility of the fees variable from private to public to enhance transparency.

2.9 Non-standard Function Naming Conventions

Severity: Info

Status: Fixed

Description :

In the ElectroSwapLockerV3 and ElectroSwapLockerV3Migrator contract, several external functions start with underscore in their names. This practice is typically reserved for internal or private functions in Solidity, can lead to confusion.

ElectroSwapLockerV3 _setFees _setTeamWallet _updateFeeWhitelist _setMigrator

ElectroSwapLockerV3Migrator _setLocker _setSourceLocker

Recommendation :

Rename these external functions to remove the leading underscore, adhering to standard naming conventions for better readability and clarity.

2.10 Redundant Address Casting of Variables.

Severity: Info

Status: Fixed

Location ElectroSwapLockerV3/ElectroSwapLockerV3.sol

Description :

In the lockPosition function of the ElectroSwapLockerV3 contract, the addresses boltAddress and msg.sender are unnecessarily typecast to the address type.

Code Affected

```
TransferHelper.safeTransferFrom(address(boltAddress), address(msg.sender), deadAddress, (
    fees.boltFlatFee * 1e18));
```

Recommendation :

Remove the unnecessary typecasting.