BLOCKAPEX

SMART CONTRACT SECURITY

V 2.0

DATE: 25th JULY 2024

PREPARED FOR: ELEKTRIK

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.

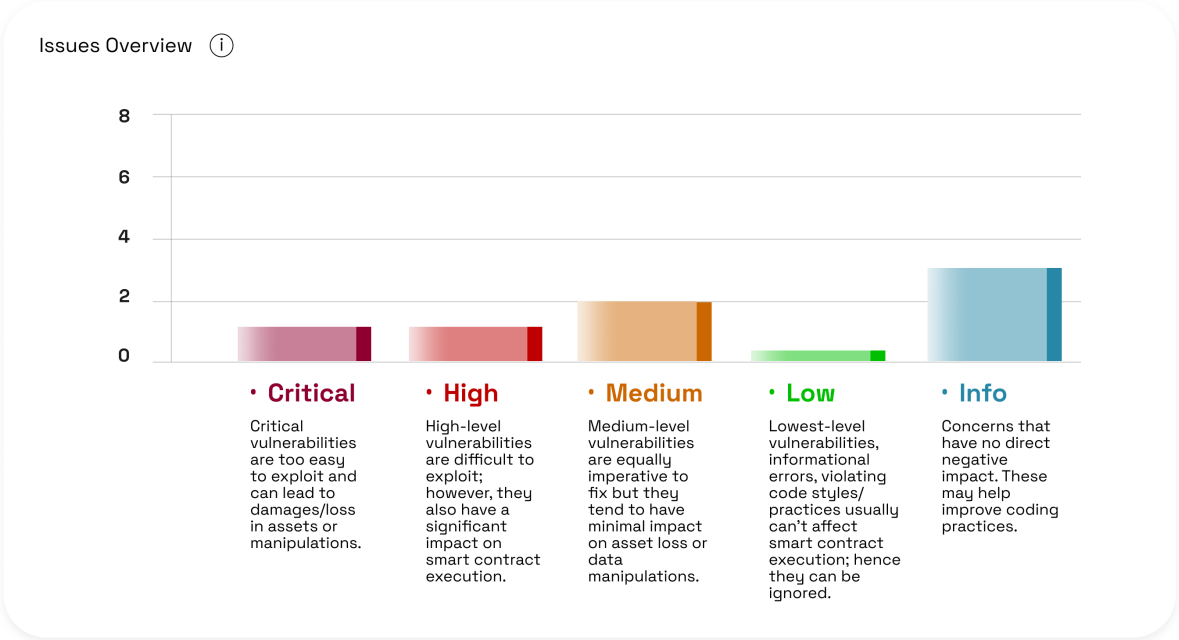
BlockApex has a focus on blockchain security, maintaining an expertise hub to navigate this dynamic field. We actively contribute to security research and openly share our findings with the community. Our work is available for review at our public repository, showcasing audit reports and insights into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

Our team conducted a Filtered Audit, engaging Three auditors to independently examine the Elektrik Staking Contracts. This rigorous approach included a meticulous line-by-line code review to identify potential vulnerabilities. Following the initial manual inspection, all flagged issues were thoroughly re-examined and re-tested to confirm their validity and ensure accuracy in our final assessment.



1.1 Scope

1.1.1 In Scope

Elektrik Staking is a forked of Velodrome Protocol with some custom tweeks in the code. Elektrik is a decentralized exchange (DEX) protocol integrated with the Lightlink Network. Elekrtik Staking contracts are part of improvement on the DEX which allows users to stake and Lock LP tokens to become part of the protocol and take decision on matters such as Pool additions and Emmision Rates on the pools. Stakers can earn ELTK for their participation in the Elektrik Protocol. The audit focused on the logic of Staking and Voting Mechanism designed to pass a voting proposal and emmision rates.

Treasury.sol: The treasury contract holds swap fees to distribute to veEltk holders (voters). It allows for the funding of specific epochs with tokens, enables users to claim rewards based on their voting power, and includes mechanisms for emergency withdrawals of funds.

Staker.sol: This contract manages liquidity staking in V3 pools, tracking user positions and liquidity. It includes functionalities for depositing, withdrawing, and updating liquidity positions, as well as calculating and distributing staking rewards. The contract integrates with a liquidity mining pool deployer and a farm booster for enhanced staking rewards.

Voter.sol: The voter contract handles governance proposals, allowing users to create, vote on, and execute proposals for adding new pools and adjusting allocations. It manages proposal lifecycles, records votes, and ensures that proposals meet the required thresholds before execution. The contract also tracks the weights of pools and total voting power within the system.

VotingEscrow.sol: The voting escrow contract allows users to lock tokens in exchange for veEltk, which gives them voting power and staking rewards. It supports creating, increasing, and withdrawing locks, with voting power decaying linearly over time. The contract includes functionalities for checkpointing, merging, and splitting locked positions, as well as delegating voting power.

Contracts in Scope: /src/*

Initial Commit Hash: 4a29dc38c813651b2f5712f484421c0e42cf4b64

Final Commit Hash: 6f650e8a26a7027032e181825d5573ef290fb16c

Project Goals: The engagement was scoped to provide a security assessment and audit of Elektrik Staking Contracts. Being a fork of velodrome attention was focused on the alterations done in the code. We sought to answer the following non-exhaustive list of questions:

1. Are access control applied correctly to all functions?
2. Are there any Reentrancy vulnerabilities in the contract ?
3. Is there proper input validation ?
4. Are events emitted properly for important functions ?
5. Are there any front-running issues in the contract ?
6. Could an attacker manipulate the voting mechanism ?
7. Can an attacker play edge scenarios in the epoch voting mechanism ?
8. Can an attacker manipulate Emission's via unauthorized means ?
9. Can attacker leverage flash loan attacks in the contract ?

1.1.2 Out of Scope

Features or functionalities not explicitly listed within the "In Scope" section, such as backend operations unrelated to the direct functioning of the smart contracts or external system integrations, are considered outside the scope of this audit.

1.2 Methodology

The codebase was audited using a filtered audit technique by a team of Three(3) auditors over a span of 3 weeks. The process began with a reconnaissance phase where the auditors developed a foundational understanding of the codebase. This initial phase helped form presumptions for the developed codebase. As the audit progressed into the manual code review phase, auditors made the Proofs of Concept (POCs) to verify their findings. This phase was designed to identify logical flaws, complemented by code optimizations, software and security design patterns, code styles, and best practices.

1.3 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.

1.4 Summary of Findings Identified

S.No	Severity	Findings	Status
#1	CRITICAL	Incorrect Epoch Validation Allows Premature Reward Claims.	FIXED
#2	HIGH	Inconsistent Voting Outcomes Due to Unanticipated Voter Participation.	FIXED
#3	MEDIUM	Manipulation of Proposal Voting through Strategic Vote Retraction.	FIXED
#4	MEDIUM	Broken Access control paradigm can lead to bricked functionality.	FIXED
#5	INFO	Unused Custom Error Definitions.	FIXED
#6	INFO	Public Functions Not Used Internally could be marked as External.	FIXED
#7	INFO	Unused Variable in Treasury.sol.	FIXED

2 Findings and Risk Analysis

2.1 Incorrect Epoch Validation Allows Premature Reward Claims.

Severity: Critical

Status: Fixed

Location :

1. `elektrik_staking_contracts/src/Treasury.sol`

Description The claim function in the Treasury contract allows users to collect rewards for specified epochs and token addresses. Currently, the function validates only the last epoch provided by the user to determine if its voting period has ended. This approach introduces a critical vulnerability where users can prematurely claim rewards for the current epoch. This occurs if the last epoch in the provided list has concluded its voting period.

```
if ((TimeLibrary.epochVoteEnd(_epoch[_epoch.length - 1]) &gt; block.timestamp)) revert
    VotingHasNotEndedYet();
```

To exploit this vulnerability, a malicious user could submit epochs in a sequence such as [current epoch, firstepoch]. If the voting period for the first epoch has ended, the function incorrectly permits the user to claim rewards for the current epoch, disregarding its ongoing voting period. This behavior allows users to exploit the system by claiming rewards before all epochs have completed their voting periods.

Impact :

- **Premature Reward Claims:** Users can claim rewards prematurely for the current epoch if earlier epochs in the list have already ended their voting periods.
- **Inconsistencies in Reward Distribution:** This vulnerability may lead to inconsistencies where rewards are distributed incorrectly based on the flawed validation logic.
- **Potential Economic Loss:** Incorrect reward distribution could lead to economic loss if rewards are claimed before their rightful time, affecting the integrity and fairness of the Treasury contract.

Proof of Concept

```
function test_treasury() public {
    uint256[] memory epoch = new uint256[](2);
    address[] memory tokenAddress = new address[](1);
    epoch[0] = block.timestamp; // current epoch
    epoch[1] = 0; // first epoch
    tokenAddress[0] = address(token);
    // Attempt to claim rewards
    treasury.claim(tokenId, epoch, tokenAddress,
address(this)); }
```

Results

```
emit Claim(voter: Treasury_test: [0x90193C961A926261B756D1E5bb255e67ff9498A1],
token: ElektrikUtility: [0xd04404bcf6d969FC0Ec22021b4736510CAcec492], reward:
10000000000000000000 [1e21], epoch: 604800 [6.048e5])
```

```
emit Claim(voter: Treasury_test: [0x90193C961A926261B756D1E5bb255e67ff9498A1],
token: ElektrikUtility: [0xd04404bcf6d969FC0Ec22021b4736510CAcec492], reward:
10000000000000000000 [1e21], epoch: 0)
```

Recommendation It is recommended to enhance the claim function to validate each epoch in the provided array. Ensure that rewards can only be claimed if all epochs have fully completed their voting periods.

2.2 Inconsistent Voting Outcomes Due to Unanticipated Voter Participation

Severity: High

Status: Fixed

Location :

1. `elektrik_staking_contracts/src/escrow/Voter.sol`

Description The current implementation of the add function in the voting contract records the voting power at the time of voting but recalculates the total voting power (_totalWeight) at the time of proposal execution. This approach can lead to failure of the execution of a proposal due to addition of new voters just before proposal expiry. Specifically, if a new voter with significant voting power is added just before the proposal expiry, the _totalWeight will increase, causing the proposal to potentially fail the threshold check and not be executed. This issue is also valid in the other direction. If voters exit the system just before the proposal expires, the _totalWeight will decrease, making the proposal highly executable even if it wouldn't have met the threshold under the original voting conditions.

Scenario 1: New Voter Added Just Before Expiry

1. There are initially 3 voters, each with 100 voting power, making the total voting power 300.
2. Only 2 voters vote for a proposal, making _poolWeight = 200.
3. Due to the continuous decrease in voting power, let's assume at expiry, the _totalWeight is 240 (for the initial voters).
4. Just before the proposal expiry, a new voter with 100 voting power is added, making _totalWeight = 340.
5. The check 200 < (340 * 2) / 3 Passes, as 200 < 226.67.

Scenario 2: Voter Exits Just Before Expiry

1. There are initially 4 voters, each with 100 voting power, making the total voting power 400.
2. Only 2 voters vote for a proposal, making _poolWeight = 200.
3. Due to the continuous decrease in voting power, let's assume at expiry, the _totalWeight is 360 (for the initial voters).
4. Just before the proposal expiry, an existing voter with 90 voting power exits, making _totalWeight = 270.
5. The check 200 < (270 * 2) / 3 Fails, as 200 < 180.

Recommendation :

- Restrict or penalize new voters from voting on proposals in the current epoch and exclude their voting power from the _totalWeight calculation.
- Additionally, if a voter's lock expiry is earlier than the proposal expiry, they should be restricted from voting.
- Impose a penalty on voters who retract their votes in the last epoch.
- Prevent voters from retracting their votes in the last hour of the voting period to avoid last-minute result changes.

2.3 Manipulation of Proposal Voting through Strategic Vote Retraction.

Severity: Medium

Status: Fixed

Location :

1. `elektrik_staking_contracts/src/escrow/Voter.sol`

Description A voter can manipulate the outcome of a proposal vote by strategically retracting their votes before the end of each epoch. If a proposal is set to last for 4 weeks (4 epochs), a voter can vote to push the proposal towards the required 66% threshold and then retract their vote before the epoch ends. This cycle can be repeated for all 4 epochs. This tactic can mislead other voters into believing the proposal has enough support to pass, causing them to refrain from voting under the assumption that the proposal will pass anyway. Ultimately, the voter retracts their votes at the last moment, flipping the results and potentially preventing the proposal from passing.

Recommendation The protocol should implement a restriction or penalty for users who retract their vote from a particular proposal. Specifically, once a user retracts their vote from a proposal, they should not be allowed to vote again on the same proposal.

2.4 Broken Access control paradigm can lead to bricked functionality

Severity: Medium

Status: Fixed

Location :

1. `elektrik_staking_contracts/src/escrow/Voter.sol`

Description Upon deployment of the Voter contract, the msg.sender is automatically added as the whitelisted address in the isWhitelisted mapping. The onlyWhitelisted modifier is applied to this mapping. Whitelisted candidates are authorized to call updateWhitelisted, setFeeTreasury, and updateProposalFee. Upon discussion with the dev team it was found that more users can and will be added in the future upon development of DAO. The issue arises because additional users can be added to the isWhitelisted mapping. If a whitelisted candidate becomes rogue, they can call updateWhitelisted and set all addresses's statuses to false, thereby disabling functionality protected by the onlyWhitelisted modifier. This includes the ability to update proposal creation fees and treasury fees address. The root cause of this issue is broken access control. A hierarchical structure should be in place where a higher authority, such as the Owner, has the exclusive right to add or remove whitelisted candidates. Candidates should not be able to alter their own or others' statuses.

Impact If a whitelisted candidate goes rogue, they can manipulate the system by setting the proposal fees to a minimum value (e.g., 1), redirecting the treasury address to their own, and subsequently removing themselves and the original msg.sender from the whitelist using updateWhitelisted(). This action would prevent any new additions to the isWhitelisted mapping, effectively bricking the functionality.

Recommendation To rectify this issue, it is recommended to revise the access control mechanism. Introduce distinct roles for the Owner and whitelisted candidates, ensuring that only the Owner has the authority to add or remove individuals from the isWhitelisted mapping. This change will prevent candidates from altering their own or others' statuses, thereby maintaining the integrity and functionality of the contract.

2.5 Unused Custom Error Definitions.

Severity: Info

Status: Fixed

Description The codebase contains multiple unused custom error definitions. These unused errors add unnecessary complexity and can cause confusion, making the code harder to maintain and understand. Moreover, even unused custom declarations can increase the bytecode size, leading to higher gas costs during deployment.

Detailed Instances Treasury.sol

- Line 21: error NotAllowed(address);
- Line 22: error NonExistentVote();
- Line 23: error InvalidPool();

IReward.sol (escrow/interfaces)

- Line 5: error InvalidReward();
- Line 6: error NotAuthorized();
- Line 7: error NotGauge();
- Line 8: error NotEscrowToken();
- Line 9: error NotSingleToken();
- Line 10: error NotVotingEscrow();
- Line 11: error NotWhitelisted();

IVoter.sol (escrow/interfaces)

- Line 27: error OutOfTimeDuration();

Recommendation Remove all identified unused custom errors from the codebase to reduce complexity, improve code maintainability, and optimize gas cost.

2.6 Public Functions Not Used Internally could be marked as External.

Severity: Info

Status: Fixed

Description Several functions in the codebase are marked as public but are not internally called within the contract. Changing these functions to external can optimize gas usage and improve contract efficiency.

Recommendation Update the visibility of identified functions from public to external to enhance gas efficiency and contract performance.

2.7 Unused Variable in Treasury.sol.

Severity: Info

Status: Fixed

Description An unused variable PRECISION is defined in the Treasury.sol contract, adding unnecessary code complexity.

Code Affected

```
1  uint256 private constant PRECISION = 10**18;
```

Recommendation Remove the unused variable to streamline the code and reduce complexity.

Disclaimer:

The smart contracts provided by the client with the purpose of security review have been thoroughly analyzed in compliance with the industrial best practices till date w.r.t. Smart Contract Weakness Classification (SWC) and Cybersecurity Vulnerabilities in smart contract code, the details of which are enclosed in this report.

This report is not an endorsement or indictment of the project or team, and they do not in any way guarantee the security of the particular object in context. This report is not considered, and should not be interpreted as an influence, on the potential economics of the token (if any), its sale, or any other aspect of the project that contributes to the protocol's public marketing.

Crypto assets/ tokens are the results of the emerging blockchain technology in the domain of decentralized finance and they carry with them high levels of technical risk and uncertainty. No report provides any warranty or representation to any third-party in any respect, including regarding the bug-free nature of code, the business model or proprietors of any such business model, and the legal compliance of any such business. No third party should rely on the reports in any way, including to make any decisions to buy or sell any token, product, service, or asset. Specifically, for the avoidance of doubt, this report does not constitute investment advice, is not intended to be relied upon as investment advice, is not an endorsement of this project or team, and is not a guarantee as to the absolute security of the project.

Smart contracts are deployed and executed on a blockchain. The platform, its programming language, and other software related to the smart contract can have vulnerabilities that can lead to hacks. The scope of our review is limited to a review of the programmable code and only the programmable code, we note, as being within the scope of our review within this report. The smart contract programming language itself remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer or any other areas beyond the programming language's compiler scope that could present security risks.

This security review cannot be considered a sufficient assessment regarding the utility and safety of the code, bug-free status, or any other statements of the contract. While BlockApex has done their best in conducting the analysis and producing this report, it is important to note that one should not rely on this report only - we recommend proceeding with several independent code security reviews and a public bug bounty program to ensure the security of smart contracts.