Leverj Staking Smart Contracts Audit Report

Preamble

This audit report was undertaken by **BlockchainLabs.nz** for the purpose of providing feedback to **Leverj Staking**.

It has subsequently been shared publicly without any express or implied warranty.

Solidity contracts were sourced from the public Github repo leverj/staking at this commit e8716e4a11881fad181b5330206d8b0c27a58510 - we would encourage all community members and token holders to make their own assessment of the contracts.

Scope

The following contracts were subject for static, dynamic and functional analyses:

- Fee.sol
- Stake.sol

Focus areas

The audit report is focused on the following key areas - though this is not an exhaustive list.

Correctness

- No correctness defects uncovered during static analysis?
- No implemented contract violations uncovered during execution?
- No other generic incorrect behaviour detected during execution?
- Adherence to adopted standards such as ERC20?

Testability

- Test coverage across all functions and events?
- Test cases for both expected behaviour and failure modes?
- Settings for easy testing of a range of parameters?
- No reliance on nested callback functions or console logs?
- Avoidance of test scenarios calling other test scenarios?

Security

- No presence of known security weaknesses?
- No funds at risk of malicious attempts to withdraw/transfer?
- No funds at risk of control fraud?
- Prevention of Integer Overflow or Underflow?

Best Practice

- Explicit labeling for the visibility of functions and state variables?
- Proper management of gas limits and nested execution?
- Latest version of the Solidity compiler?

Analysis

- Work Paper

Issues

Severity Description

Minor	A defect that does not have a material impact on the contract execution and is likely to be subjective.

Moderate	A defect that could impact the desired outcome of the contract execution in a specific scenario.
Major	A defect that impacts the desired outcome of the contract execution or introduces a weakness that may be exploited.
Critical	A defect that presents a significant security vulnerability or failure of the contract across a range of scenarios.

Minor

- **Prefer explicit declaration of variable types** - `Best practice` It is recommended to explicitly define your variable types, this confirms your intent and safeguards against a future when the default type changes. e.g `uint256` is preferred to `uint` even though they are the same type. Example #L38 View on GitHub

  - [ ] Fix not required

- **Tokens should emit a generation event on creation** - `Best practice` When a token is first created it should log a `Transfer` event from address `0x0`. This is useful for tools such as EtherScan.io so they can see tokens have been minted. (Some more info here) #L39 View on GitHub

  - [x] Fixed 9e5ab6b7209d05a2ce141fd47ccd896754c5bf33

Moderate

- **Using old compiler version** - `Best practice` This version of the solidity compiler is very old, you should use the most recent stable branch. You should also use a consistent compiler between all contracts. The `pragma solidty...` line should be at the top of the file, before any `import` s happen. I'm logging this as a moderate issue because an actual exploit coming from a bug in an old compiler is rare, but the consequences could be severe if something was ever discovered. View on GitHub
  - [x] Fixed 9e5ab6b7209d05a2ce141fd47ccd896754c5bf33

Major

- None found

Critical

- None found

Observations

- Consider adding an event for when an `operator` is set in `Owned.sol`.
- `Staking.sol:L47` asks about the difference in storage cost between types, I have created a Test Contract which shows that the method you used is the cheapest. The difference between each type is negligible and there is a difference of around 44 gas between the cheapest and most expensive.
- Fee.sol can mint new tokens, so some protection should be made against overflowing.
- Consider adding a ClaimTokens function to Stake.sol so that you can retrieve any random tokens people send to the contract, they would be stuck otherwise. Ensure that you require that the token address you claim is not Lev tokens or Fee tokens so that your users can trust that it won't be used for malicious purposes. Rough example of this idea.

Conclusion

We are statisfied that these Smart Contracts do not exhibit any known security vulnerabilities. Overall the code is well written and the developers have been responsive throughout the audit process. The contracts show care taken by the developers to follow best practices and are well documented. There is high test coverage which should increase confidence in the security of these contracts, and their maintainability in the future. As part of our auditting process we added some new tests to improve the coverage.

Disclaimer

Our team uses our current understanding of the best practises for Solidity and Smart Contracts. Development in Solidity and for Blockchain is an emergering area of software engineering which still has a lot of room to grow, hence our current understanding of best practise may not find all of the issues in this code and design.

We have not analysed any of the assembly code generated by the Solidity compiler. We have not verified the deployment process and configurations of the contracts. We have only analysed the code outlined in the scope. We have not verified any of the claims made by any of the organisations behind this code.

Security audits do not warrant bug-free code. We encourge all users interacting with smart contract code to continue to analyse and inform themselves of any risks before interacting with any smart contracts.