Element Finance Council Smart Contracts Review

By: ChainSafe Systems

October 2021

Element Finance Council
Smart Contracts Review

Auditor: Oleksii Matiiasevych, Tanya Bushenyova, Anderson Lee

WARRANTY

This Code Review is provided on an “as is” basis, without warranty of any kind, express or implied. It is not intended to provide legal advice, and any information, assessments, summaries, or recommendations are provided only for convenience (each, and collectively a “recommendation”). Recommendations are not intended to be comprehensive or applicable in all situations.
ChainSafe Systems does not guarantee that the Code Review will identify all instances of security vulnerabilities or other related issues.

1. Introduction

Element Finance requested ChainSafe Systems to perform a review of the Council smart contracts. The contracts can be identified by the following git commit hash:

```
a1de6a966818fe3cd42c386dfe9058d2707a75bf
```

There are 16 contracts in scope.

After the initial review, Element Finance team applied a number of updates which can be identified by the following git commit hash:

```
4003f1bf818eec73cba545d49ecf1f897aa0f203
```

Additional verification was performed after that.

2. Disclaimer

The review makes no statements or warranties about utility of the code, safety of the code, suitability of the business model, regulatory regime for the business model, or any other statements about fitness of the contracts for any specific purpose, or their bug free status. The review documentation below is for internal management discussion purposes only and should not be used or relied upon by external parties without the express written consent of ChainSafe Systems.

3. Executive Summary

All the initially identified high severity issues were fixed and are not present in the final version of the contract, except for one frontrunning issue which is planned to be mitigated with an after transaction check.

There are **no** known compiler bugs for the specified compiler version (0.8.3), that might affect the contracts' logic.

There were 1 critical, 5 major, 7 minor, 37 informational/optimizational issues identified in the initial version of the contracts. The non-informational issues found in Council contracts were not present in the final version of the contracts, and a single new informational issue was discovered. They are described below for historical purposes. The one remaining major issue allows temporarily gaining victims voting power during the initial LockingVault deposit. See 5.13 for more details.

We have enjoyed working with the Element Finance team on this project.

4. Line-by-Line Review

4.1. OptimisticGrants, line 83: Note, outdated comment regarding `_recipient.`

4.2. Treasury, line 17: Optimization, `_governance` should be immutable.

4.3. History, line 79: Note, `‘data < uint256(1) << 192’` statement could be made more readable with `‘data <= type(uint192).max’`.

4.4. VestingVaultStorage, line 3: Note, `Storage` import is not used.

4.5. GSCVault, line 23: Note, instead of `‘60 * 60 * 24 * 4’` use `‘4 days’` for readability.

4.6. GSCVault, line 79: **Major**, `queryVotePower()` allows querying the same vault multiple times gaining unfair voting power.

4.7. GSCVault, line 94: Optimization, no need to rewrite the `joined` variable, just update `votingVaults`.

4.8. LockingVault, line 141: Minor, wrong order of params in `VoteChange` event.

4.9. LockingVault, line 163: Minor, wrong order of params in `VoteChange` event.

4.10. LockingVault, line 186: Minor, wrong order of params in `VoteChange` event.

4.11. LockingVault, line 192: Minor, wrong order of params in `VoteChange` event.

4.12. LockingVault, line 160: Note: the comments are not relevant (copy of comments from deposit function).

4.13. LockingVault, line 162: Note: the comments are not relevant (copy of comments from deposit function).

4.14. LockingVault, line 185: Note: the comments are not relevant (copy of comments from deposit function).

4.15. OptimisticRewards, line 27: Note, instead of `‘60 * 60 * 24 * 7’` user `‘7 days’` for readability.

4.16. VestingVault, line 145: Note, no need to redeclare the `_delegatee` variable.

4.17. VestingVault, line 205: Minor, `grant.delegatee` is read from storage after the `grant` being deleted which results in the 0x0 address being emitted instead of the actual delegatee

4.18. VestingVault, line 207: Minor, `grant.latestVotingPower` is read from storage after the `grant` being deleted which results in the 0 voting power being emitted instead of the actual voting power.

4.19. VestingVault, line 236: **Critical**, infinite voting power inflation if old and new delegates are the same.

4.20. CoreVoting, line 131: Minor, anyone could spam by creating empty proposals as it doesn't require to have any voting power at all.

4.21. CoreVoting, line 213: **Major**, voting in the same block as proposal creation allows one to use a flashloan to buy a ton of votes to immediately pass the proposal.

4.22. CoreVoting, line 249: **Major**, calculating proposal hash using an abi.encodePacked allows an attacker to invalidate proposals with hash collisions.

Example:

```
abi.encodePacked(
[“0x5B38Da6a701c568545dCfcB03FcB875f56beddC4”],[“0x”])
== abi.encodePacked(
[“0x5B38Da6a701c568545dCfcB03FcB875f56beddC4”, “0x0000000000000000000
000000000000000000020”, “0x0000000000000000000000000000000000000001”]
,[])
```

4.23. CoreVoting, line 267: **Major**, if the call requires more than 2M gas, it could intentionally fail to censor the proposal. In order to do that, the executor will have to specify a little bit less gas than needed.

5. Line-by-Line Verification. Remaining and Acknowleged Issues

5.1. Spender, line 18: Optimization, would be cheaper to use a struct of `{lastSpendBlock, total}` to not pollute the storage, instead of storing spends for every block in `blockExpenditure`.

5.2. Spender, line 103: Optimization, having `uint256[3] calldata` instead of `uint256[] memory` will make the `setLimits` function cheaper to execute.

5.3. Treasury, line 38: Note, `.transfer()` will fail for recipients with heavy fallbacks.

5.4. Authorizable, line 10: Optimization, `authorized` mapping could be made internal to make deployment cheaper.

5.5. Authorizable, line 51: Note, consider checking that `who` is not 0x0 address to avoid accidentally losing ownership. Another way would be to use a 2 step ownership transfer process.

5.6. History, line 166: Note, `_find()` will revert if `length == 0`.

5.7. History, line 226: Note, `length - 1` reverts if `length == 0`.

5.8. MerkleRewards, line 17: Optimization, `lockingVault` variable should be made immutable to make claiming cheaper.

5.9. MerkleRewards, line 32: Optimization, use local `_lockingVault` variable instead of reading it from storage.

5.10. VestingVaultStorage, line 21: Optimization, `created` and `expiration` fields could be made `uint48` and fit in a single slot with `delegatee` field.

5.11. GSCVault, line 61: Note: `assert` could be replaced with `require` with an info message. Also, a check that `votingVaults.length == extraData.length` could be added to validate the input.

5.12. GSCVault, line 160: Optimization: In the `queryVotingPower` function `members[msg.sender].joined` is read from storage multiple times. It would be cheaper to read it once, assign its value to a local variable and use it in these cases.

5.13. LockingVault, line 116: **Major**, frontrunning is possible to temporarily gain votes. Could be mitigated with an after transaction check. Attack scenario: someone makes an initial deposit, frontrunner makes deposit for them setting themself as a delegate, now the victim transaction gets mined and all votes go to the attacker.

5.14. LockingVault, line 150: Note, truncation of `amount`. Only a problem if the contract could hold more than `uint96` tokens on balance.

5.15. OptimisticRewards, line 67: Note, if new root excludes old recipients, then `GSCVault` will never be able to kick those who proved themselves with the old root.

5.16. VestingVault, line 42: Note, make sure that `initialize` function is called in the same transaction as contract deployment, or perform a post deployment check to confirm proper setup to avoid a frontran initialization.

5.17. VestingVault, line 76: Note, `_manager()` function could return `.data` for simplicity.

5.18. VestingVault, line 83: Note, `_timelock()` function could return `.data` for simplicity.

5.19. VestingVault, line 96: Note, `_unvestedMultiplier()` function could return `.data` for simplicity.

5.20. CoreVoting, line 173: Optimization, excessively reading empty `votingPower` field.

5.21. CoreVoting, line 237: Optimization, `votingPower` is read twice from storage.

5.22. CoreVoting, line 264: Optimization, `unlock` is read twice from storage.

5.23. CoreVoting, line 352: Note, could use `bytes4(_calldata)` instead of assembly for better readability.

5.24. SimpleProxy, line 54: Optimization, should be cheaper to use `calldatasize()` everywhere instead of the `calldataLength` variable on a stack.

5.25. SimpleProxy, line 58: Note, if an implementation has a fallback function it will never be reached.

5.26. SimpleProxy, line 75: Optimization, comments are correct, there is no need to update the free memory pointer if the whole function is executed in assembly.

5.27. SimpleProxy, line 81: Optimization, there is no point to clear the garbage even if it is there, `delegatecall` will ignore extra bytes anyway.

5.28. SimpleProxy, line 100: Optimization, should be cheaper to use `returndatasize()` everywhere instead of the `returndataLength` variable on a stack.

Oleksii Matiiasevych

Tanya Bushenyova

Anderson Lee