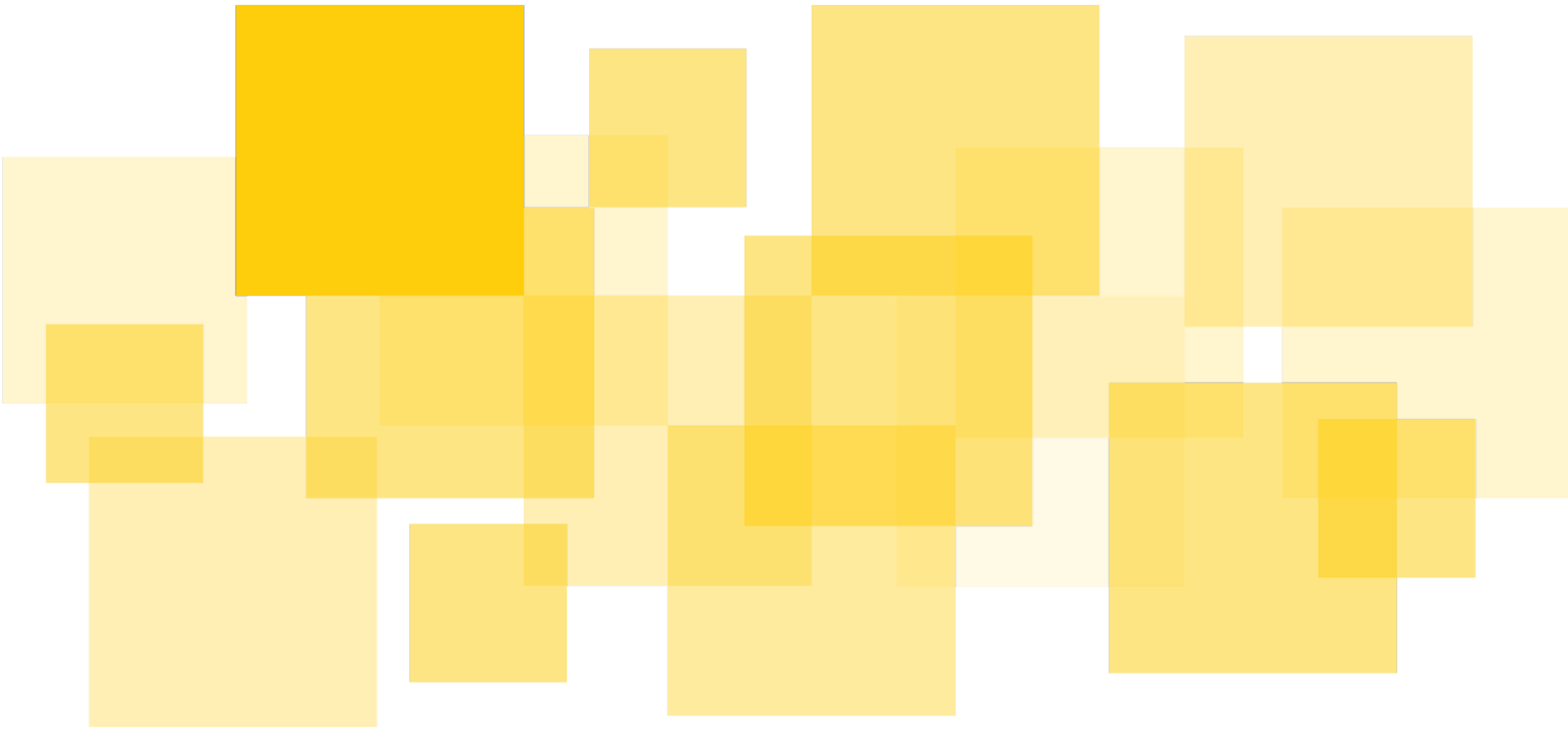
Security Audit Report

Element Finance Governance Contract

Delivered: October 15th, 2021



Prepared for Element Finance by

runtime verification

Summary

Runtime Verification, Inc. conducted a security audit on the Element Finance governance smart contracts. The audit was conducted by Daejun Park and Raoul Schaffranek from August 16, 2021 to September 17, 2021, and from October 4 to October 15, 2021.

Several issues have been identified as follows:

- Implementation flaws: A02, A08, A11, A12, B09, B20
- Potential security vulnerabilities: A03, A04, A05, A10, A14, A16, A17, A18
- Fault tolerance concerns: A01, A06, A07, A09, A13, A15, B10

A number of additional suggestions have also been made, including:

- Input validations: B06, B07, B13, B16, B17, B18
- Best practices: B01, B02, B08, B11, B12, B14, B15, B19
- Gas optimization: B02, B03
- Code readability: B04, B05

All the critical security issues have been addressed by the client. Details can be found in the Findings section as well as the Informative Findings section.

The code is well written and thoughtfully designed, following best practices.

We note that the sheer number of findings does *not* imply that the security and quality of the code are unsatisfactory, but more of results from thorough audits and partly due to the inherent complexity of the contracts for good reason.

Scope

The target of the audit is the smart contract source files at git-commit-id 3d751c9 as well as code fixes made up to 6ad7765.

The audit focused on the following core contracts, reviewing their functional correctness:

- CoreVoting.sol
- simpleProxy.sol
- vaults/GSCVault.sol
- vaults/LockingVault.sol
- vaults/OptimisticRewards.sol
- vaults/VestingVault.sol
- features/Airdrop.sol
- features/OptimisticGrants.sol

- features/Spender.sol
- features/Timelock.sol
- features/Treasury.sol
- libraries/Authorizable.sol
- libraries/History.sol
- libraries/MerkleRewards.sol
- libraries/Storage.sol
- libraries/VestingVaultStorage.sol

The audit is limited in scope within the boundary of the Solidity contract only. Off-chain and client-side portions of the codebase as well as deployment and upgrade scripts are *not* in the scope of this engagement, but are assumed to be correct (see below).

Assumptions

The audit is based on the following assumptions and trust model.

- External token contracts conform to the ERC20 standard, especially that they *must* revert in failures. Also, they do *not* allow any callbacks (e.g., ERC721 or ERC777) or any external contract calls. Moreover, the total supply of tokens must *not* exceed the limit implicitly set by the contracts.[1]
- For the authorizable contracts, their owners and authorized users are trusted to behave correctly and honestly, as described in the Security Model section.
- The contracts will be deployed and integrated correctly, as described in the Contracts Deployment section.

Methodology

Although the manual code review cannot guarantee to find all possible security vulnerabilities as mentioned in Disclaimer, we have followed the following approaches to make our audit as thorough as possible. First, we rigorously reasoned about the business logic of the contract, validating security-critical properties to ensure the absence of loopholes in the business logic and/or inconsistency between the logic and the implementation. Then, we carefully checked if the code is vulnerable to known security issues and attack vectors. Finally, we symbolically executed part of the compiled bytecode to systematically search for unexpected, possibly exploitable, behaviors at the bytecode level, that are due to EVM quirks or Solidity compiler bugs.

[1] Currently, the limit is "2^96 / 10^(#decimals)". For example, for a token with 18 decimals, the total supply limit is around 79 billions.

Security Model

The governance contracts are operated by multiple parties, and the secure operation of governance requires all the participating agents to behave correctly, honestly, and diligently.

Voters (i.e., stackers or their delegates):

- Voters must check the validity of proposals, and vote for them only if they are correct and benign.
- Critical proposals, e.g., ones for updating system parameters, or spending significant amounts of funds, must be time-locked and go through an extra challenging period.
- Great care is needed when the targets of proposed transactions are untrusted external accounts. If the accounts are a contract account, their code should be audited before approving the proposals, to ensure that they are not malicious, and prevent any potential reentrancy attacks.[2] Extra caution is also needed if the target is an empty account (i.e., an account whose nonce is zero and code is empty), because of the potential code injection attack.[3]
- For proposals that update system parameters, voters must check that the proposals include necessary resync processes, so that the system status will be adjusted to be consistent with the updated parameters. (For example, see A06, A07, A13, and B10 for resync requirements.)
- Voters must check that new proposals do *not* conflict with pending proposals or "zombie" proposals (i.e., proposals that were approved but failed in execution, and not yet expired). (For example, see A17.)
- Voters must have a good understanding of the system behaviors when interacting with it. (For example, see A01, A12, and B05.)

Governance steering committee (GSC) members:

- As privileged agents for most parts of the system, the GSC must be trusted, and *never* compromised.
- In addition to being correct and benign, the GSC must diligently and proactively protect the system, by challenging malicious proposals, and proposing revoking proposals, etc.
- The GSC should continuously monitor the system status and propose repairs if needed, e.g., in case of A10 or A15.

[2] Note that if the contract is upgradable, code injection attacks are possible, as described in A14.
[3] The target address might have been precomputed by CREATE2, and adversaries may front-run the CREATE2 code deployment before the proposal execution, or execute both atomically by themselves since anyone can execute approved proposals.

OptimisticRewards proposers and revokers:

- Proposers must maintain the off-chain Merkle tree correctly and honestly.
- Revokers must verify proposed Merkle roots against the off-chain tree, and revoke if not.
- They must verify the validity of the tree, e.g., pair-sortedness, and leaf-uniqueness (A09).

VestingVault managers:

- Managers must award grants and manage funds correctly and honestly.

Contracts Deployment

The initial status of deployed contracts should be as follows. Deployed instances (i.e., addresses) are written in *italic*.

Governance entities

Governance: the first instance of the `CoreVoting` contract

- `owner`: *Timelock*
- `authorized`: { *GSC* }
- `approvedVaults`: { *LockingVault*, *VestingVault*, *OptimisticRewards* }

GSC: the second instance of the `CoreVoting` contract

- `owner`: *Timelock*
- `authorized`: { }
- `approvedVaults`: { *GSCVault* }

Voting vaults

LockingVault: the first instance of the `SimpleProxy` contract (upgradable)

- `proxyGovernance`: *Timelock*
- `proxyImplementation`: the only instance of the `LockingVault` contract
- `token`: *GovernanceToken*

VestingVault: the second instance of the `SimpleProxy` contract (upgradable)

- `proxyGovernance`: *Timelock*
- `proxyImplementation`: the only instance of the `VestingVault` contract
- `token`: *GovernanceToken*
- `manager`: *VestingManager*
- `timelock`: *Timelock*

OptimisticRewards: the only instance of the `OptimisticRewards` contract

- `owner`: *Governance*
- `authorized`: { *GSC* }
- `proposer`: *RewardsProposer*
- `lockingVault`: *LockingVault*
- `token`: *GovernanceToken*

GSCVault: the only instance of `GSCVault` contract

- `owner`: *Timelock*
- `authorized`: {}
- `coreVoting`: *Governance*

Features

Timelock: the only instance of the `Timelock` contract

- `owner`: *Governance*
- `authorized`: { *GSC* }

Treasury: the only instance of the `Treasury` contract

- `owner`: *Timelock*
- `authorized`: {}

Spender: the only instance of the `Spender` contract

- `owner`: *Timelock*
- `authorized`: { *Governance* }
- `token`: *GovernanceToken*

OptimisticGrants: an instance of the `OptimisticGrants` contract

- `_governance`: *Governance*
- `token`: *GovernanceToken*

Airdrop: the only instance of the `Airdrop` contract

- `owner`: *Timelock*
- `authorized`: {}
- `lockingVault`: *LockingVault*
- `token`: *GovernanceToken*

Tokens

GovernanceToken: an instance of the `ERC20PermitWithMint` contract

- `owner`: *Governance*

Agents

VestingManager: an EOA or a multisig account

RewardsProposer: an EOA or a multisig account

Remarks

There exist two instances of the `CoreVoting` contract: *Governance* and *GSC*.

Only *LockingVault* and *VestingVault* are upgradable, and all the others are immutable.

There may exist multiple instances of the `OptimisticGrants` contract, but only one of them will be used as part of the system.

Disclaimer

This report does not constitute legal or investment advice. The preparers of this report present it as an informational exercise documenting the due diligence involved in the secure development of the target contract only, and make no material claims or guarantees concerning the contract's operation post-deployment. The preparers of this report assume no liability for any and all potential consequences of the deployment or use of this contract.

Smart contracts are still a nascent software arena, and their deployment and public offering carries substantial risk. This report makes no claims that its analysis is fully comprehensive, and recommends always seeking multiple opinions and audits.

This report is also not comprehensive in scope, excluding a number of components critical to the correct operation of this system.

The possibility of human error in the manual review process is very real, and we recommend seeking multiple independent opinions on any claims which impact a large quantity of funds.

Findings

A01: Condition for proposal approval

[Severity: Unknown | Difficulty: N/A | Category: Security]

The approval of a proposal does *not* require the majority YES votes but merely to have more YES votes than NO votes.

Scenario

Suppose a proposal gets only a *single* YES vote, but zero NO vote and many MAYBE votes. This proposal can be executed as long as the number of MAYBE votes is greater than or equal to the required quorum size.

Recommendation

It is recommended to introduce an additional requirement of a certain minimum size of YES votes for the proposal approval.

Status

Acknowledged by the client. (See Issue 14.)

A02: Event ProposalExecuted

[Severity: N/A | Difficulty: N/A | Category: Functional Correctness]

The `ProposalExecuted` event can emit true even if the given proposal is *not* executed.

```
        emit ProposalExecuted(proposalId, results[0] > results[1]);
```

CoreVoting.sol#L226

Recommendation

Add the extra condition for the proposal approval.

Status

Fixed in PR 15.

A03: Non-atomic batch execution

[Severity: High | Difficulty: N/A | Category: Security]

The batch of calls in proposals is executed non-atomically. That is, the execution will terminate normally and also delete the proposal, even if some of the calls fail or even no calls succeed.

Recommendation

It is recommended to have the execute() function to revert if one of the batch calls fails, to ensure the atomic execution of batches.

Discussion

If the non-atomic execution is a desired feature, then it is recommended to support both atomic and non-atomic execution modes, and have users specify the intended mode of execution. This will help prevent users from making a mistake due to ignorance.

Status

Fixed in PR 35.

A04: Validation of proposal hashes

[Severity: High | Difficulty: Medium | Category: Security]

In general, `abi.encodePacked()` is ambiguous when their arguments are dynamic, and thus is *not* recommended for the purpose of integrity checks such as the following:

```
// ensure the data matches the hash
require(
      keccak256(abi.encodePacked(targets, abi.encode(calldatas))) ==
            proposals[proposalId].proposalHash,
      "hash mismatch"
);
```

CoreVoting.sol#L246-L251

Specifically, `abi.encodePacked()` omits the size information of the given arrays. Thus, in theory, it is possible to pass the above hash check with a hand-crafted `targets` calldata whose length field is different than the original (while all other fields are the same with the original).

Scenario

If the length field of `targets` is smaller than the original, then execute() will execute only the calls to the `targets` up to the given length. A more problematic case is when the length field is bigger than the original, where the original `targets` array used in proposal() can contain hidden elements that can bypass the quorum requirement.

Recommendation

Use `abi.encode()` instead of `abi.encodePacked()`.

Status

Fixed in PR 28.

A05: Input validation in proposal()

[Severity: Unknown | Difficulty: Low | Category: Input Validation]

Calling proposal() with the empty `targets` array always succeeds and increases `proposalCount` (as well as adding a dummy proposal).

Scenario

A malicious user can repeat calling proposal() with the empty array to increase `proposalCount` to an arbitrary value, which might be used as part of bigger attacks.

Recommendation

It is recommended to require the non-emptiness of `targets` in proposal().

Status

Fixed in PR 28.

A06: Changes of approved vaults during voting period

[Severity: High | Difficulty: High | Category: Security]

The vote counting process does *not* take into account potential changes of the list of associated vaults in the middle of the voting period. The vote tally that has already been counted will never be recounted even if the vault list is changed in the middle.

Scenario

Suppose a voting vault has been compromised and it grants a large voting power to adversaries, and they voted for malicious proposals using the compromised voting power. Suppose the governance realizes the situation during the voting period for the malicious proposals, and thus removes the compromised vault from the approved list. However, the adversaries' votes that have already been made with the compromised vault will never be revoked, even if the vault is removed from the list. The only way to prevent the approval of the proposals is that sufficiently many benign users vote against them. However, suppose that the remaining voting period is too short for that, and it ends before that happens. Then, the malicious proposals are approved and can be executed immediately.

Recommendation

Redesign the architecture to handle properly the change of approved vaults.

Status

Acknowledged by the client.

A07: Asynchronous updates of voting power in VestingVault

[Severity: Unknown | Difficulty: Medium | Category: Integration]

In VestingVault, some changes in voting powers are *not* immediately reflected in the voting power record, until users explicitly synchronize them. This causes inconsistency between the actual voting power and the recorded value. This inconsistency biases the count of votes, and thus may lead to incorrect voting results.

Scenarios

1. If unvestedMultiplier is changed by the governance, the actual voting power for some users will be immediately changed accordingly, but the recorded value for them will stay the same until the explicit synchronization calls are made. No atomicity guarantee is made at the code level for updating unvestedMultiplier and synchronizing the record. This temporary inconsistency might be exploited in certain bigger attacks.
2. In case that unvestedMultiplier is set less than 100%, the actual voting power continues to increase over time, but the record value will remain unchanged, being almost always smaller than the actual value, until the explicit synchronization. The inherent gap between the continuously changing value and the recorded value that is updated only on demand, might be abused in one way or another.

Recommendation

For the short term, clearly document this limitation and/or implement the frontend of governance to ensure the atomic update socially or systematically. This will mitigate the issue described in the first scenario, but not much in the second.

For the longer term, improve the design for the case of “unvestedMultiplier < 100%.”

Status

Acknowledged by the client.

A08: Self delegation in VestingVault

[Severity: High | Difficulty: Low | Category: Security]

VestingVault.delegate() doesn't work correctly if `_to` is equal to `grant.delegatee`.

Scenario

If adversaries delegate on themselves, then they can increase their voting power (almost) double in the record.

Recommendation

Fix the logic to properly handle the self delegation case, or simply reject the self delegation.

Status

Fixed in PR 30.

A09: Duplicated leaves in MerkleRewards trees

[Severity: Medium | Difficulty: High | Category: Functional Correctness]

MerkleRewards may *not* work correctly if the associated Merkle tree contains multiple leaves for the same user.

Scenario 1

Suppose the proposer of OptimisticRewards commits a root of a Merkle tree that contains two leaves for Alice, one for 10 tokens and another for 5 tokens, accidentally or maliciously. Suppose Alice has already claimed the 10 tokens from the first leaf. Now Alice attempts to claim the 5 tokens from the second leaf, but it fails to pass the following check, where `claimed[Alice]` is 10, `amount` is 5, but `totalGrant` is only 5 (which comes from the second leaf).

```
require(claimed[msg.sender] + amount <= totalGrant, "Claimed too much");
```

MerkleRewards.sol#L92

Scenario 2

Suppose Alice is a GSC member whose voting power mostly comes from the OptimisticRewards vault. Then Bob, a malicious proposer of OptimisticRewards can revoke her membership as follows. Bob adds another leaf node, *N* = (user = Alice, totalGrant = 0), to the Merkle tree, and proposes the new Merkle root. Suppose the proposed new root is not challenged and becomes a new reward root. Then, Bob calls GSCVault.kick() for Alice with the Merkle proof for the new leaf node *N*. Then, Alice's voting power from the OptimisticRewards vault is counted as zero, and if Alice's voting power from other vaults is not large enough, she will be kicked out from the committee, even if she actually has enough voting power.

Recommendation

For the short term, clearly document the requirement of the uniqueness of leaves in the tree, so that the reward proposer does *not* make a mistake of adding multiple leaves for the same user.

For the long term, improve the logic to properly handle the case of duplicated leaves.

Status

Acknowledged by the client.

A10: Potential flash loan vulnerability

[Severity: High | Difficulty: High | Category: Security]

When a proposal is created, it gets the YES vote that amounts to the *current* voting power of the proposer. This allows malicious proposers to use a flash loan to temporarily increase the voting power and use it to vote for their proposals.

Scenario

Suppose Alice takes a flash loan more than a half of the total governance token supply, deposits it into the locking vault, and proposes a malicious proposal. Then, no one can prevent the malicious proposal from being executed after the locking period passes.

Recommendation

For the short term, add a protection measure to make the flash loan attack harder.

For the long term, consider using the average value in a certain period to compute the voting power at the time of voting. (See the discussion below.)

Status

Addressed in PR 32 by taking the previous block's balance in computing the voting power.

Discussion

Taking the previous block's balance may prevent the flash loan based attacks, but may not be effective for attacks using non-flash loans (but very short-term, possibly non-collateralized, loans over only a couple of blocks). In general, the voting power based on the balance at a specific moment is more prone to manipulation than that based on the average balance over a longer period of time. And, indeed, if someone's balance is kept high enough during a sufficient period of time, then he should have a true voting power, regardless of his intention being malicious or not.

A11: Upgradable proxy pattern for VestingVault

[Severity: High | Difficulty: Low | Category: Integration]

While VestingVault employs the upgradable proxy pattern, it misses a storage initialization function.

Scenario

VestingVault cannot be properly initialized once it is created.

Recommendation

Implement the initialization function.

Status

Fixed in PR 33.

A12: Underflow in History.sol

[Severity: Low | Difficulty: Low | Category: Usability]

The `History._find()` function reverts when the given `length` parameter is 0, due to the underflow in the following:

```
        uint256 maxIndex = length - 1;
```

History.sol#L224

This underflow leads to `History.find()` or `History.findAndClear()` reverting because the `storageMapping[who]` array is empty and thus the `History._find()` function is called with the `length` argument being zero, triggering the underflow.

Scenario

Suppose Alice calls vote() with LockingVault and VestingVault being the voting vaults, while her voting power record in VestingVault is still empty. Then the vote() function will always fail even if she has sufficient voting power in LockingVault.

Recommendation

Fix the function to return zero when the length is zero to improve the user experience.

Status

Addressed in PR 40 by *explicitly* reverting when the length is zero.

A13: Validity of voting period

[Severity: Low | Difficulty: High | Category: Functional Correctness]

The voting period (i.e., "`lockDuration + extraVoteTime`" in CoreVoting) is required to be smaller than the period for which the historical voting power is recorded (i.e., the `staleBlockLag` value of all the associated vaults.) However, this requirement is *not* enforced by the code.

Scenario

Suppose the governance makes a mistake of increasing the value of either lockDuration or extraVoteTime too much and the condition "`lockDuration + extraVoteTime < staleBlockLag`" is violated for a certain vault. Then, vote() may fail while it shouldn't, because queryVotePower() may fail due to the historical voting power record having not sufficiently covered the entire voting period.

Recommendation

For the short term, clearly document the requirement so that the governance does not make such a mistake.

For the long term, refactor the code to ensure the condition when `lockDuration` or `extraVoteTime` is updated.

Status

Acknowledged by the client.

A14: Reentrancy vulnerability

[Severity: High | Difficulty: Medium to High | Category: Security]

There exist reentrancy vulnerabilities.

Scenario

Suppose that a proposal of sending funds to a contract owned by Alice is approved. Then Alice can execute the approved proposal multiple times, by performing the following in a single transaction:

1. Alice upgrades the target contract so that it can reenter the CoreVoting.execute() or Timelock.execute() function upon receiving funds.
2. Alice calls CoreVoting.execute() or Timelock.execute() to execute the approved proposal.

Recommendation

Employ reentrancy protection measures. For example:

- Use a mutex for the execute() function, e.g., ReentrancyGuard.
- Delete the proposal (or the callTimestamps mapping entry for Timelock) before making external calls.
- Have a proposal to include a gas limit, and execute the proposal with the given limit.

Status

Fixed in PR 42.

A15: Griefing attacks for high-profile delegates

[Severity: Low | Difficulty: High | Category: Security]

`History._clear()` iterates over the given array, where the array can be increased arbitrarily large by adversaries, causing the function to revert due to exceeding the block gas limit. This can be exploited for griefing certain voters, especially high-profile delegates, to delay approval for certain proposals, which can be used as part of bigger attacks.

Scenario

Suppose adversaries repeat depositing a tiny amount to LockingVault in every block, for several days, with all of them being delegated to Alice. Then Alice's storageData array size will increase by tens of thousands after several days. Later, when it's time to delete the tens of thousands elements, Alice may no longer be able to vote indefinitely, because `queryVotePower()` (in turn, `findAndClear()` and `_clear()`) would consume more gas than the block limit.

Recommendation

Improve the logic of the `_clear()` function so that it can partially delete the given entries.

Status

Acknowledged by the client.

A16: Double execution vulnerability in Timelock

[Severity: High | Difficulty: Medium to High | Category: Security]

A batch of calls may be doubly executed in Timelock.

Scenario

Consider the following two scenarios where increaseTime() is executed after execute() for the same `callHash`:

1. The authorized GSC contract *accidentally* or *maliciously* calls increaseTime() for the `callHash` that has already been executed. *[Difficulty: High]*
2. *OR*, the GSC *legitimately* calls increaseTime() near the end of the original waiting period, but increaseTime() is delayed for some reason until the original waiting period passes, and immediately adversaries front-run execute() before the increaseTime() transaction. *[Difficulty: Medium]*

Then, `callTimestamps[callHash]` becomes non-zero again, and the subsequent execute() may lead to double execution of the same batch of calls.

Recommendation

Add an extra condition "`require(callTimestamps[callHash] != 0);`" in the `increaseTime()` function to ensure that the given `callHash` is active.

Status

Fixed in PR 48.

A17: Differentiation of identical Timelock proposals

[Severity: High | Difficulty: High | Category: Security]

The Timelock contract may not properly work when multiple identical Timelock proposals are approved in a short time period.

Scenario 1

Suppose the governance approves a registerCall(X) proposal. Soon after, a problem is found for X, and the governance approves a stopCall(X) proposal. Then, two proposals are unlocked at similar time, but stopCall(X) is accidentally or maliciously executed before registerCall(X). Then, the execution of both proposals will "succeed", but X will not be canceled but registered. To cancel X, the governance must go through another voting process to re-approve a stopCall(X) proposal, which may not be completed on time, and X may slip through.

Scenario 2

Suppose the governance approves a registerCall(Y) proposal. Later, a problem is found for Y, and a stopCall(Y) proposal is created. Suppose, however, that it takes too long to approve stopCall(Y), and Y has already slipped through by the time when stopCall(Y) is approved. Suppose that no one executes the approved stopCall(Y) because it has no effect. Long later, however, suppose another registerCall(Y) proposal is approved and executed for good reason, but then adversaries can execute the zombie stopCall(Y) to revoke Y, which is undesired.

Recommendation

For the short term, add a check to ensure that callTimestamps[callHash] is zero for registerCall(), and it is non-zero for stopCall().[4]

For the long term, add a counter (similar to the proposalCount in CoreVoting) to distinguish multiple identical callHashes of Timelock.

Status

Addressed in PR 49 by adopting the short-term fix.

[4] The short term fix can address only Scenario 1, but not Scenario 2. Indeed, it will worsen Scenario 2.

A18: Unchecked generic calls in Treasury

[Severity: High | Difficulty: N/A | Category: Security]

Treasury provides the genericCall() function that can be called as part of a batch of calls for the CoreVoting.exeucte() or Timelock.execute() function. While *both* the execute() functions check the status of sub-calls and revert in case of failures in any of the sub-calls, the Treasury.genericCall() does not check the status, thus no failure will propagate to the caller.

Scenario

Suppose the governance approves a proposal that involves spending the Treasury funds, say trading some tokens of Treasury in a decentralized exchange, via Treasury.genericCall(). Suppose the execute() function in CoreVoting or Timelock is called with a batch of calls that include the Treasury.genericCall(). Suppose that the Treasury token trading fails (e.g., due to the slippage). However, the genericCall() will *not* check the status, and normally return. The execute() function will continue to execute the remaining calls in the batch, which may lead to an undesired result, due to the incomplete execution of the batch.

Recommendation

Check the status of the sub-call in Treasury.genericCall(), to be consistent with the other meta-tx executors, CoreVoting.exeucte() and Timelock.execute().

Status

Fixed in PR 49.

Informative Findings

B01: Mapping quorums

[Severity: N/A | Difficulty: N/A | Category: Code Readability]

The following mapping returns 0 for default, while the default quorum size is non-zero.

```
    // mapping of address and selector to quorum
    mapping(address => mapping(bytes4 => uint256)) public quorums;
```

CoreVoting.sol#L20

Recommendation

To improve code readability, it is recommended to make the mapping private and then add a modified version of the default getter which returns the default quorum size instead of zero.

Status

Fixed in PR 16.

B02: Recycling garbage proposals

[Severity: N/A | Difficulty: N/A | Category: Gas Cost]

Unapproved proposals are never deleted, while it is considered a better practice in general to delete storage slots that are no longer necessary.

Scenario

Suppose a proposal hasn't got enough votes and the voting period has already ended. It cannot get more votes and thus never be approved in the future. However, there exists no way to delete such a garbage proposal.

Recommendation

Implement the logic to recycle garbage proposals in either execute() or a separate new function.

Discussion

Although the gas refund is reduced to almost a third, the amount is still non-negligible. Moreover, each proposal occupies five storage slots, and thus the benefit from the refund by deleting garbage proposals should *not* be underestimated.

Status

Acknowledged by the client.

B03: Packed storage update in LockingVault

[Severity: N/A | Difficulty: N/A | Category: Gas Cost]

The following two storage assignments are *not* packed into a single storage update. The current Solidity compiler optimization does *not* support such a complex assignment pattern.

```
            // TODO - Ensure that this compiles to one sstore
            // Set the delegation
            userData.who = delegate;
            // Now we increase the user's balance
            userData.amount += uint96(amount);
```

LockingVault.sol#L125-L129

Scenario

When users deposit to LockingVault for the first time, they may need to pay slightly more gas fees than necessary.

Recommendation

Refactor the code to ensure the packed assignment, e.g., as follows:

```
            uint96 newAmount = userData.amount + uint96(amount);
            userData.who = delegate;
            userData.amount = newAmount;
```

NOTE: the gas saving must be weighted against the reduced code readability to properly evaluate the benefit of this change.

Status

Acknowledged by the client.

B04: Code readability in History.sol

[Severity: N/A | Difficulty: N/A | Category: Code Readability]

The following code can be simplified for code readability.

```
uint256 mid = maxIndex + minIndex - (minIndex + maxIndex) / 2;
```

History.sol#L238

Specifically, it can be simplified as follows:

```
uint256 mid = (minIndex + maxIndex + 1) / 2;
```

Discussion

Although the two expressions are equivalent mathematically (even considering the truncation division semantics), the two don't agree in the EVM execution when "`minIndex + maxIndex`" is equal to the max uint256 value (which is $2^{256} - 1$); that is, that the original code succeeds in computing the median value, while the simplified code reverts. This trade-off between readability and robustness needs to be considered before adopting the proposed simplification.

Status

Fixed in PR 40.

B05: Cancellation of votes

[Severity: Low | Difficulty: Low | Category: Usability]

Calling vote() with the empty list of vaults will effectively cancel the previous vote, if any. It is not intuitive to expect the vote cancellation feature from vote().

Recommendation

It is recommended to revert when the given list is empty, and have a separate entrypoint function that cancels votes.

Status

Acknowledged by the client.

B06: Input validation in VestingVault

[Severity: Medium | Difficulty: High | Category: Input Validation]

The following sanity checks are desired, but missing:

- For addGrantAndDelegate(): `require(_cliff <= _expiration);`
- For changeUnvestedMultiplier(): `require(_multiplier <= 100);`

Recommendation

Add the proposed checks.

Status

Fixed in PR 30.

B07: Input validation in Timelock

[Severity: N/A | Difficulty: N/A | Category: Input Validation]

It is recommended to have the following sanity checks:

- For Timelock.execute(): `require(targets.length == calldatas.length);`

Status

Fixed in PR 40.

B08: Systematic sanity checks for explicit type conversions

[Severity: N/A | Difficulty: N/A | Category: Best Practice]

It is a defensive programming practice to systematically add a sanity check for every explicit type conversion to a smaller-sized type, unless it is *logically* clear that the value is sufficiently small. Note that in Solidity, the explicit conversion does *not* check the size, but simply truncates the higher bits, which may lead to serious problems if the given value exceeds the limit of the target type.

Status

Acknowledged by the client.

B09: Unsafe type conversions

[Severity: N/A | Difficulty: N/A | Category: Functional Correctness]

The following code snippets perform an explicit type conversion from `uint128` to `int256`, but in an unsafe way. If the input value is larger than or equal to `2^127`, then the converted value will be incorrect.

```
emit VoteChange(grant.delegatee, _who, int256(int128(newVotingPower)));
```

VestingVault.sol#L190

```
-1 * int256(int128(grant.latestVotingPower))
```

VestingVault.sol#L224

```
-1 * int256(int128(grant.latestVotingPower))
```

VestingVault.sol#L268

```
int256(newVotingPower) - int256(int128(_grant.latestVotingPower));
```

VestingVault.sol#L329

Recommendation

Replace the unsafe conversion, `int256(int128(X))`, with the safe one, `int256(uint256(X))` which has no precision loss for any value within the range of `uint128`.

Status

Fixed in PR 49.

B10: Redetermination of membership in GSCVault

[Severity: N/A | Difficulty: N/A | Category: Functional Correctness]

The GSCVault contract provides setter functions that can modify the value of storage variables that are used as a criteria to determine membership of GSC. Thus, if the storage variables are updated by the setters, then the membership of the existing committee members needs to be re-determined based on the updated criteria. However, the setter functions do *not* enforce the redetermination at the code level. Also, no documentation about this requirement can be found.

Scenario

Suppose the owner of GSCVault wants to tighten the requirements of GSC membership, and executes setCoreVoting(), setVotePowerBound(), and/or setIdleDuration() to update the criteria. However, existing committee members who no longer satisfy the requirements will still stay and can continue to vote for critical proposals, until they are explicitly kicked out by executing kick() against them.

Recommendation

For the short term, clearly document the redetermination requirements for the setter functions. That is, the execution of the setter functions must be followed by the execution of kick() against every existing member of the GSC.

For the long term, redesign the contract to enforce the redetermination process at the code level.

Status

Acknowledged by the client.

B11: Inconsistent owner initialization in CoreVoting

[Severity: N/A | Difficulty: N/A | Category: Best Practice]

It is a better practice to use the setOwner() function provided by the Authorizable library, rather than the following explicit assignment. Note that all the other contracts that inherit Authorizable use the setOwner() function, but only the CoreVoting contract doesn't, which is inconsistent with the rest of the codebase.

```
        owner = address(_timelock);
```

CoreVoting.sol#L118

Status

Fixed in PR 49.

B12: Inconsistent Solidity version requirements

[Severity: N/A | Difficulty: N/A | Category: Best Practice]

More than one version (pragma) of Solidity, especially versions 0.8.0 and 0.8.3, are mixedly used throughout the codebase. It is a better practice to unify the compiler version. Since the version 0.8.3 fixes the keccak caching bug, it can be considered to upgrade lower versions, even if the current codebase is not specifically affected by the bug.

Status

Fixed in PR 49.

B13: Sanity check for proposal IDs

[Severity: N/A | Difficulty: N/A | Category: Input Validation]

It is recommended to have an explicit check "`require(proposalID < proposalCount);`" for both vote() and execute() in CoreVoting.sol. This helps prevent error propagation in case of storage corruption.

Status

Acknowledged by the client.

B14: Inconsistent storage cleanup in Timelock

[Severity: N/A | Difficulty: N/A | Category: Code Readability]

It is recommended to also delete `timeIncreases[callHash]` whenever `callTimestamps[callHash]` is deleted (in both execute() and stopCall()). It improves the code readability and gas efficiency. See also A14 for the related issue, which needs to be addressed together with this.

Status

Fixed in PR 48.

B15: Code duplication in History.sol

[Severity: N/A | Difficulty: N/A | Category: Code Refactoring]

The find() and findAndClear() functions duplicate a code block that can be factored out to avoid the code duplication.

Status

Acknowledged by the client.

B16: Input validation for delegatee addresses

[Severity: N/A | Difficulty: N/A | Category: Input Validation]

The NatSpec documentation for both LockingVault.deposit() and MerkleRewards.claimAndDelegate() gives a clear warning against delegating to the zero address. It is recommended to enforce this at the code level, rather than relying on only the warning message.

Recommendation

Add *either* or *both* of the following checks, depending on the desired user experience:

- For LockingVault.deposit(), especially for the case of "`delegate == address(0)`":
  - `require(firstDelegation != address(0));`
- For MerkleRewards.claimAndDelegate():
  - `require(delegate != address(0));`

Furthermore, for LockingVault.deposit() in case of "`delegate != address(0)`", consider adding the following extra sanity check:[5]

- `require(firstDelegation == address(0) || firstDelegation == delegate);`

Status

Fixed in PR 49.

[5] We note that this helps early detection of honest mistakes, but may increase the surface of the griefing attack mentioned in the NatSpec document. The trade-off needs to be considered to evaluate this recommendation's benefits.

B17: Sanity checks for Merkle proofs

[Severity: N/A | Difficulty: N/A | Category: Input Validation]

As a defensive programming practice, it can be considered to have an extra check to ensure that the size of the given `merkleProof` array is equal to the Merkle tree height, in `MerkleRewards._validateWithdraw()`.

Discussion

This extra check requires an additional storage slot for storing the tree height, incurring gas overheads. Moreover, the hash collision probability would be still low even over different-sized proofs. Thus, the trade-off and cost-effectiveness need to be considered when evaluating this recommendation's benefits.

Status

Acknowledged by the client.

B18: Sanity checks for History.sol

[Severity: N/A | Difficulty: N/A | Category: Input Validation]

Considering the importance of the fundamental data structure correctness, it is recommended to add extra sanity checks as a preventative measure for potential storage corruption and/or catastrophic failures due to hidden bugs, as follows:

- For the _find() function:
  - require(staleBlock < blocknumber);
  - require(startingMinIndex < length);
- For the _clear() function:
  - require(oldMin < newMin);
  - require(newMin < length); *(possibly at the call sites)*
- For the _loadAndUnpack() function:
  - require(i < length); *(possibly at the call sites)*
- For the _setBounds() function:
  - require(minIndex < length);

Status

Fixed in PR 49.

B19: Potential inconsistency of interfaces

[Severity: N/A | Difficulty: N/A | Category: Best Practice]

Both the ICoreVoting and ILockingVault interfaces are *not* inherited by the implementation contracts, which may cause them to be outdated by mistake when the implementation contracts are updated in the future. It is a better practice to associate interfaces with their implementation so that the compiler can detect any inconsistencies that could be introduced in the future.

Status

Fixed in PR 49.

B20: Typo in function names

[Severity: N/A | Difficulty: N/A | Category: Typo]

According to the IVotingVault interface, the following function name should have been `queryVotePower` rather than `queryVotingPower`.

```
        function queryVotingPower(
```

GSCVault.sol#L145

Status

Fixed in PR 49.