

Produced for



by



Contents

1	Executive Summary	3
2	Assessment Overview	5
3	Limitations and use of report	11
4	Terminology	12
5	Findings	13
6	Resolved Findings	15
7	Notes	26

1 Executive Summary

Dear Snapshot team,

Thank you for trusting us to help Snapshot with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of Snapshot X according to [Scope](#) to support you in forming an opinion on their security risks.

Snapshot implements a configurable voting protocol for systems with decentralized governance. It allows users to create proposals which can then be voted on and potentially executed. There are a variety of contracts which allow the system to choose which users can vote, which can create proposals, how the votes are counted, and how the proposals are executed.

The most critical subjects covered in our audit are functional correctness, access control, trustworthiness and reentrancies. Several issues regarding these topics have been remedied. Access control is handled correctly throughout. Potential reentrancy vulnerabilities have been addressed. The risk of an implementation contract SELFDESTRUCT was addressed. Several risk-free issues have been acknowledged and are by design, see [Systemic bias towards accepting proposals](#), [Proposal can be updated just before voting starts](#), [Same proposal status for queued, executed or vetoed proposals](#).

The general subjects covered are upgradeability, gas efficiency and documentation. Security regarding these subjects is high. Some steps were taken to improve gas efficiency, which overall is decent. The level of documentation is satisfactory, however, some peculiarities highlighted in the [Notes](#) section could be more explicitly documented.

In summary, we find that the codebase provides a high level of security.

It is important to note that security audits are time-boxed and cannot uncover all vulnerabilities. They complement but don't replace other vital measures to secure a project.

The following sections will give an overview of the system, our methodology, the issues uncovered and how they have been addressed. We are happy to receive questions and feedback to improve our service.

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

Below we provide a brief numerical overview of the findings and how they have been addressed.

-Severity Findings	0
-Severity Findings	1
•	1
-Severity Findings	7
•	6
•	1
-Severity Findings	18
•	15
•	1
•	2

2 Assessment Overview

In this section, we briefly describe the overall structure and scope of the engagement, including the code commit which is referenced throughout this report.

2.1 Scope

The assessment was performed on the source code files inside the Snapshot X repository based on the documentation files. The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	19 May 2023	d43240f17edeaef2ae8c67367d09754d31063	Initial Version
2	20 Jun 2023	9f5f1fe49bb10c3b63f15ffd892ba0a578d67a43	Version 2
3	03 Jul 2023	6c7830f9466ad4f6324afa6b4e02fc3818791162	Version 3

For the solidity smart contracts, the compiler version 0.8.18 was chosen.

The following files were in scope:

- ProxyFactory.sol
- Space.sol
- types.sol
- authenticators/Authenticator.sol
- authenticators/EthSigAuthenticator.sol
- authenticators/EthTxAuthenticator.sol
- authenticators/VanillaAuthenticator.sol
- execution-strategies/AvatarExecutionStrategy.sol
- execution-strategies/EmergencyQuorumStrategy.sol
- execution-strategies/OptimisticQuorumExecutionStrategy.sol
- execution-strategies/SimpleQuorumExecutionStrategy.sol
- execution-strategies/VanillaExecutionStrategy.sol
- execution-strategies/timelocks/CompTimelockCompatibleExecutionStrategy.sol
- execution-strategies/timelocks/TimelockExecutionStrategy.sol
- execution-strategies/timelocks/OptimisticCompTimelockCompatibleExecutionStrategy.sol
- execution-strategies/timelocks/OptimisticTimelockExecutionStrategy.sol
- proposal-validation-strategies/ActiveProposalsLimiterProposalValidationStrategy.sol
- proposal-validation-strategies/PropositionPowerAndActiveProposalsLimiterValidationStrategy.sol
- proposal-validation-strategies/PropositionPowerProposalValidationStrategy.sol
- proposal-validation-strategies/VanillaProposalValidationStrategy.sol
- utils/ActiveProposalsLimiter.sol
- utils/BitPacker.sol
- utils/PropositionPower.sol

- `utils/SignatureVerifier.sol`
- `utils/SpaceManager.sol`
- `utils/SXHash.sol`
- `utils/SXUtils.sol`
- `utils/TimestampResolver.sol` (removed in later versions)
- `voting-strategies/CompVotingStrategy.sol`
- `voting-strategies/OZVotesVotingStrategy.sol`
- `voting-strategies/VanillaVotingStrategy.sol`
- `voting-strategies/WhitelistVotingStrategy.sol`

2.1.1 Excluded from scope

Third-party dependencies, testing files, and any other files not listed above are outside the scope of this review.

2.2 System Overview

This system overview describes the initially received version () of the contracts as defined in the [Assessment Overview](#).

Furthermore, in the findings section, we have added a version icon to each of the findings to increase the readability of the report.

Snapshot introduces Snapshot X, a framework for on-chain governance. The framework is designed to be modular, flexible, and upgradeable, making it capable of adapting to a wide range of governance use-cases. It includes several components to handle proposal creation, voting, and execution, all of which work together to form a robust governance mechanism.

2.3 System architecture

Every Snapshot X governance system revolves around a main component called *Space*. DAOs that implement their governance mechanism through Snapshot X deploy their *Space*, which coordinates the governance mechanism. Governance operates through the creation of proposals, their voting, and their execution. Users interact with a *Space* through authenticators, which ensure that the caller has the right to perform the desired operation.

A proposal consists of an execution strategy and a payload. Users create proposals through the `Space propose` function. Every *Space* is configured with a Proposal validation strategy, an external contract which enforces rules on who is allowed to create new proposals. After a proposal is created, a waiting time of `votingDelay` allows members of the DAO to properly evaluate the proposal, before voting starts, and the proposal creator can update the proposal during this period. After the voting delay has passed, users can vote *For*, *Against*, or *Abstain*, on a proposal, through the `vote` function of the *Space*. The voting power of a user is evaluated through one or many voting strategies, which are external contracts configured in the *Space*. The voting lasts between `minVotingDuration` and `maxVotingDuration` seconds. It can finish early if an early quorum is reached. If the proposal is accepted, the proposal can be executed. Execution of a proposal is triggered by a user calling the `execute` function of the *Space*. The execution strategy of the proposal is then called, with the proposal payload as argument. The execution strategy is an external contract, which validates whether a proposal has passed, and defines how the proposal is executed. Generally, execution strategies hold special rights on the systems controlled by the DAO, and are allowed to perform privileged operations on them. A

Space has a controller, which can call a few privileged operations on the Space. These operations are `cancel()`, which cancels a proposal which has not yet been executed, `upgradeTo()`, which allows to upgrade the implementation contract for a Space deployed through an upgradeable proxy, and `updateSettings()`, which changes the configuration on a Space. The configuration of the Space is defined by the following parameters:

1. `votingDelay`, the time before voting starts after a proposal is created.
2. `minVotingDuration`, the minimum duration of a voting period.
3. `maxVotingDuration`, the maximum duration of a voting period.
4. `proposalValidationStrategy`, the contract that is queried to check if a user can create a proposal.
5. `votingStrategies`, the contracts that can be used by a voter to collect their voting power.
6. `authenticators`, the contracts that are allowed to call the Space's `vote()`, `propose()`, and `execute()` functions on behalf of users.

A proposal, once created, can be in one of the following states:

1. *VotingDelay*: the vote is not open yet, the proposer can edit the proposal.
2. *VotingPeriod*: the vote is ongoing.
3. *VotingPeriodAccepted*: the `minVotingDuration` has elapsed, and the vote is accepted although voting is still open.
4. *Accepted*: `maxVotingDuration` has elapsed, the vote is closed and it has reached the acceptance conditions defined by the execution strategy.
5. *Executed*: the `execute` method of the execution strategy has successfully been called for the proposal.
6. *Rejected*: The voting period is finished, and the acceptance conditions defined by the execution strategy have not been met.
7. *Cancelled*: The owner of the Space has cancelled the proposal before its execution.

2.4 Authenticators

Users of a Space do not directly call the Space `vote()`, `updateProposal()`, and `propose()` functions. Instead, the Space only allows a small set of *Authenticator* contracts to call these functions, which accept the proposal `author` or `voter` as first argument, on behalf of users. This indirection layer allows the creation of new trusted ways of authenticating votes. Currently, two authenticator contracts are implemented: *EthSigAuthenticator* and *EthTxAuthenticator*.

2.4.1 EthTxAuthenticator

EthTxAuthenticator implements the trivial authentication mechanism of forwarding a call to the Space only if the `msg.sender` matches the `author` or `voter` address.

2.4.2 EthSigAuthenticator

EthSigAuthenticator implements EIP-712 signature verification to call the Space functions on behalf of users that have cryptographically signed a digest of their voting or proposal creation intentions but are not themselves directly interacting with the blockchain.

2.5 Proposal Validation Strategies

Proposal creation in a Space is controlled by a *Proposal Validation Strategy*. This is an external contract that is queried to validate if the author of a proposal is allowed to create a new proposal in the Space. Configuration parameters for the proposal validation strategies are stored in the Space. The same Proposal Validation Strategy contracts can be shared between different Spaces. Three Proposal Validation Strategies are currently implemented: *ActiveProposalsLimiterProposalValidationStrategy*, *PropositionPowerProposalValidationStrategy*, and *PropositionPowerAndActiveProposalsLimiterValidationStrategy*.

2.5.1 *ActiveProposalsLimiterProposalValidationStrategy*

This proposal validation strategy implements a counter that keeps track of the number of proposals that a user has created, only allowing new proposals if the counter is below a certain threshold. If the user has created no new proposals after a cooldown period since the last creation, the counter is reset to zero. This contract can be shared between several Spaces. Space segregation is performed, so the counters of different Spaces do not interact.

2.5.2 *PropositionPowerProposalValidationStrategy*

This proposal validation strategy queries a list of configured voting strategies to gather the voting power of the proposal creator. If the voting power is above a certain threshold, the proposal creation is allowed.

2.5.3 *Combination of the previous two*

The *PropositionPowerAndActiveProposalsLimiterValidationStrategy* proposal validation strategy combines the logic of the other two, accepting a proposal creation only if the active proposals are few enough, and the voting power of the proposer is beyond a threshold.

2.6 Voting Strategies

A Space configures one or more *Voting Strategies*. Voting Strategies gather the voting power of a voter at a certain point in time through the `getVotingPower` function. They are queried when the `vote` function is called, and in proposal validation strategies that validate the proposer's voting power. Several can be active in a Space at the same time, and the voter calls `vote()` with a list of indices as argument. These indices specify which of the up to 255 voting strategies in a Space should be used to gather their voting power. A voter can specify fewer than the active ones in the Space if they want to save gas by not querying voting strategies where they do not hold voting power. Three voting strategies are implemented: *CompVotingStrategy*, *OZVotesVotingStrategy*, and *WhitelistVotingStrategy*.

2.6.1 *CompVotingStrategy*

CompVotingStrategy queries the voting power of a user through the `getPriorVotes()` interface provided by Compound-like governance tokens. `getPriorVotes()` allows to query the governance token balance at a given block number. Since Snapshot X operates with timestamp instead of block numbers, a translation layer between timestamp and block numbers is required, which is implemented through the abstract contract *TimestampResolver.sol*

2.6.2 *OZVotesVotingStrategy*

OZVotesVotingStrategy implements the same logic as *CompVotingStrategy*, but queries a token which implements the `getPastVotes()` interface, which is what is implemented by governance tokens in OpenZeppelin.

2.6.3 *WhitelistVotingStrategy*

WhitelistVotingStrategy retrieves the voting power of a user from a list stored in the space. This list holds voting powers for specific users, sorted by user address. Binary search is used to find a specific address in this list. The Space controller can update the whitelist.

2.7 Execution Strategies

A Space delegates vote counting and proposal execution to *Execution Strategies*. A proposal creator can define an arbitrary *Execution Strategy* on a new proposal; however, most proposals will be executed through execution strategies that hold special powers on behalf of the DAO. Execution Strategies define the `getProposalStatus` function, which receives the proposal parameters and the vote amounts as arguments and returns the current status of the proposal. Three abstract strategies currently define different vote counting methodologies: *SimpleQuorumExecutionStrategy*, *OptimisticQuorumExecutionStrategy*, and *EmergencyQuorumStrategy*.

2.7.1 Quorum Types

2.7.1.1 *SimpleQuorumExecutionStrategy*

This abstract contract accepts a proposal if its total vote count exceeds a predefined quorum, and the *For* votes are more than the *Against* votes. Between the `minEndTimestamp` and `maxEndTimestamp` of a proposal, it is still possible to vote on the proposal, but it is in the `VotingPeriodAccepted` state if more *For* votes than *Against* exist, and the quorum has been reached. This makes the proposal executable before the end of the voting period.

2.7.1.2 *EmergencyQuorumStrategy*

This abstract contract defines a `getProposalStatus` function that can accept a proposal before the end of the voting period if the total number of votes is bigger than an emergency quorum and the *For* votes exceed the *Against* votes. No concrete contract currently derives from this execution strategy, but new execution strategies could be created in the future to derive from it.

2.7.1.3 *OptimisticQuorumExecutionStrategy*

This abstract execution strategy considers any proposal to be accepted, unless the *Against* votes exceed a quorum. It allows proposal validation with less gas spent, since it supposes that proposers are not malicious, and most proposals will be accepted. No concrete contract currently derives from this execution strategy.

2.7.2 Execution Mechanics

Beyond the role of the execution strategy in vote counting, the role of the execution strategy is to execute a proposal when it passes a vote. The proposal execution is expected to act on behalf of the DAO and perform privileged actions over which the execution strategy is authorized to do. How a proposal payload is interpreted depends on the chosen execution strategy, however, in general it will consist of a list of transactions which will be sent from a privileged account. There are three concrete execution strategies which implement this transaction relaying mechanism on behalf of an accepted proposal: *AvatarExecutionStrategy*, *CompTimelockCompatibleExecutionStrategy*, and *TimelockExecutionStrategy*. The execution of the `execute()` method of the execution strategies is limited to trusted Spaces, since it allows calling arbitrary addresses with admin powers.

2.7.2.1 *AvatarExecutionStrategy*

AvatarExecutionStrategy executes a successful proposal according to *SimpleQuorumExecutionStrategy* by forwarding the list of transactions defined in the proposal payload to a Gnosis-Safe or similar contract, which will in turn execute the transactions.

2.7.2.2 *CompTimelockCompatibleExecutionStrategy*

CompTimelockCompatibleExecutionStrategy queues accepted proposals in a Compound Governance timelock. This timelock will allow the execution strategy to execute queued proposals once the timelock delay has elapsed. Beyond the guarded `execute()` method, which only the Space can call, this execution strategy implements an unpermissioned `executeQueuedProposal` method which allows anybody to ultimately execute a proposal which had been previously queued. A privileged role called `vetoGuardian` has the power to cancel queued proposals.

2.7.2.3 *TimelockExecutionStrategy*

This execution strategy behaves similarly to *CompTimelockCompatibleExecutionStrategy*. However, it implements the timelock itself. It has the `vetoGuardian` role to cancel queued transactions and can act as a treasury for a DAO. Its `execute` method queues lists of transactions which will be executed either with `CALL` or `DELEGATECALL`.

The execution strategies are expected to be deployed as proxies to concrete implementation contracts.

2.7.2.4 *OptimisticCompTimelockCompatibleExecutionStrategy*

Added in `1.10.0`, this strategy behaves identically to the *CompTimelockCompatibleExecutionStrategy*, except that it inherits from the *OptimisticQuorumExecutionStrategy*.

2.7.2.5 *OptimisticTimelockExecutionStrategy*

Added in `1.10.0`, this strategy behaves identically to the *TimelockExecutionStrategy*, except that it inherits from the *OptimisticQuorumExecutionStrategy*.

2.8 Proxy Factory

Several contracts can be deployed as proxies to implementation contracts. The proxy factory allows deployment at predictable addresses and initialization in a single transaction. The Space contract can be deployed as an upgradeable proxy since the contract implements `OpenZeppelinUUPSUpgradeable`. The execution strategies can be deployed as non-upgradeable proxies.

2.9 Trust Model

Snapshot X allows implementation of arbitrary governance mechanisms. Fully decentralized governance can be implemented with self-ownership of all the privileged contracts. Alternatively, more centralized approaches could be chosen, which would allow privileged EOAs or a multi-sig to perform potentially disruptive actions on the DAO governance. It is the role of the deployer of a Snapshot X Space to properly configure the system to implement trustless governance. In particular, we advise caution with who is set as the owner of the Space contract and the execution strategy contracts. The owner of the Space contract can change governance parameters and arbitrarily upgrade the contract. The owner of an execution strategy can define which Space is allowed to call the execution strategy, effectively providing the owner arbitrary execution powers on behalf of the DAO.

The veto guardian in the timelock execution strategies can also be self-owned by the DAO, by setting it as another execution strategy which doesn't have an execution delay.

The choice of parameters in a Space can greatly affect the mechanisms under which DAO governance operates, setting low quorum threshold, low voting delay, or low durations will enable a proposal creator to pass proposals in their favor. Likewise, active voting strategies, proposal validation strategies, and authenticators must be carefully inspected to match the desired governance rules.

Misconfigurations are possible, we expect a helpful deployment mechanism to allow deployers to avoid them, and we assume that the deployers for any specific DAO are not malicious.

3 Limitations and use of report

Security assessments cannot uncover all existing vulnerabilities; even an assessment in which no vulnerabilities are found is not a guarantee of a secure system. However, code assessments enable the discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary. In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it. Some of the issues may affect the entire application, while some lack protection only in certain areas. This is why we carry out a source code assessment aimed at determining all locations that need to be fixed. Within the customer-determined time frame, ChainSecurity has performed an assessment in order to discover as many vulnerabilities as possible.

The focus of our assessment was limited to the code parts defined in the engagement letter. We assessed whether the project follows the provided specifications. These assessments are based on the provided threat model and trust assumptions. We draw attention to the fact that due to inherent limitations in any software development process and software product, an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third-party infrastructure which adds further inherent risks as we rely on the correct execution of the included third-party technology stack itself. Report readers should also take into account that over the life cycle of any software, changes to the product itself or to the environment in which it is operated can have an impact leading to operational behaviors other than those initially determined in the business specification.

4 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- *Likelihood* represents the likelihood of a finding to be triggered or exploited in practice
- *Impact* specifies the technical and business-related consequences of a finding
- *Severity* is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

Likelihood	Impact		
	High	Medium	Low
High			
Medium			
Low			

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

5 Findings

In this section, we describe any open findings. Findings that have been resolved have been moved to the [Resolved Findings](#) section. The findings are split into these different categories:

- : Related to vulnerabilities that could be exploited by malicious actors
- : Architectural shortcomings and design inefficiencies
- : Mismatches between specification and implementation
- : Violations to the least privilege principle

Below we provide a numerical overview of the identified findings, split up by their severity.

-Severity Findings	0
--------------------	---

-Severity Findings	0
--------------------	---

-Severity Findings	1
--------------------	---

- [Systemic Bias Towards Accepting Proposals](#)

-Severity Findings	2
--------------------	---

- [Proposal Can Be Updated Just Before Voting Starts](#)
- [Same Proposal Status for Queued, Executed or Vetoed Proposals](#)

5.1 Systemic Bias Towards Accepting Proposals

CS-SNAPSHOT-005

The voting system implemented by the various execution strategies is biased towards accepting proposals. This is due to the *VotingAccepted* status of a proposal, where if a proposal receives enough support between the `minVotingDuration` and `maxVotingDuration`, it can be executed even if users are still voting on it. Therefore, a proposal needs to only have enough support for a very brief moment during this period to be executed. In contrast, for the proposal to be rejected it must not receive enough support for this entire duration. Hence, there is a systemic bias towards accepting "narrowly" supported proposals over rejecting them.

Acknowledged:

Snapshot states:

We are happy with that design. If a DAO does not want the bias, they can set `minVotingDuration` and `maxVotingDuration` to be equal, thereby removing the *VotingPeriodAccepted* status. In our docs we will clarify this point so that users are aware of the potential bias.

5.2 Proposal Can Be Updated Just Before Voting Starts

CS-SNAPSHOT-019

The author of a proposal is able to change the execution strategy and payload of their proposal up to the moment when voting starts. This could mean that voters inform themselves about the proposal, decide how they would like to vote, and then vote when the voting period begins. However, if the author of the proposal updates it just before the start of the voting period, the voters may not realize it. This is especially problematic if the duration of the voting period is short, or if there is an incentive to vote quickly (for example in the case of an emergency).

Acknowledged:

Snapshot states:

We don't think this needs to be addressed at the protocol level but will provide notifications at the UI level to minimize the probability of confusion.

5.3 Same Proposal Status for Queued, Executed or Vetoed Proposals

CS-SNAPSHOT-021

In `TimelockExecutionStrategy` and `CompTimelockCompatibleExecutionStrategy`, executing or vetoing a transaction has the same observable effect of setting the `executionTime` of the payload to 0. Different events (`ProposalVetoed` vs `ProposalExecuted`) are emitted, but the proposal status as reported by `getProposalStatus` will in both cases be `Executed`. An on-chain actor will have no way to tell if a queued proposal has been executed or vetoed.

Acknowledged:

Snapshot states:

From the POV of the space, the proposal has been executed if the `execute` function gets called without reverting. This is the case even if the proposal then later gets vetoed. We therefore think the current setup is fine.

6 Resolved Findings

Here, we list findings that have been resolved during the course of the engagement. Their categories are explained in the [Findings](#) section.

Below we provide a numerical overview of the identified findings, split up by their severity.

-Severity Findings	0
--------------------	---

-Severity Findings	1
--------------------	---

- [Unfair Voting Power Timestamp](#)

-Severity Findings	6
--------------------	---

- [ActiveProposalsLimiter Doesn't Handle maxActiveProposals Decreases Correctly](#)
- [Proxy Deployment Vulnerable to Front-Running](#)
- [Space Relies Heavily on Execution Strategy for Correctness](#)
- [TimelockExecutionStrategy Deployed as Proxy at Risk From Implementation SELFDESTRUCT](#)
- [Timestamp Resolver May Return Incorrect Block Number](#)
- [_quorumReached Counts votesAgainst Toward the Quorum](#)

-Severity Findings	16
--------------------	----

- [ActiveProposalsLimiter User Data Could Use a Struct](#)
- [Better Storage and Struct Packing Possible](#)
- [Comp Timelock Doesn't Support Duplicate MetaTransaction Queueing](#)
- [Function Parameter Location Optimizations](#)
- [Function Visibility Can Be Restricted](#)
- [Inconsistency in Quorum Modifiability](#)
- [Inefficient Boolean Mappings](#)
- [Multiple Voting if Voting Strategy Calls Untrusted Code](#)
- [No Setter for Delay in TimelockExecutionStrategy](#)
- [Potential Reentrancy in executeQueuedProposal Method](#)
- [Quorum Not Included in the AvatarExecutionStrategySetUp Event Emission](#)
- [TimelockExecutionStrategy Not Fully ERC165 Compliant](#)
- [Unnecessary Binary Search](#)
- [Unused Function](#)
- [Various Reentrancy Possibilities During Proposal Execution](#)
- [Voting Power Not Easily Estimated in User Interface](#)

6.1 Unfair Voting Power Timestamp



The `snapshotTimestamp` of a proposal is the time at which the proposal was created. It is used as a reference timestamp to decide how much voting power a user should have for certain voting strategies, namely the *CompVotingStrategy* and *OZVotesVotingStrategy*. More specifically, this timestamp resolves to the block prior to the block in which the proposal was created. This leads to two problems:

1. The proposal author, who knows exactly what block is relevant to their proposal, can purchase voting power for only that block, then sell it again. This gives them an easy way to have a large influence on their proposal, without other users getting the same opportunity.
2. Other users cannot "prepare" for the vote - even if it's a proposal that they are heavily invested in, their voting power is already decided when the proposal is created.

Instead, letting the timestamp at which the voting period starts decide the voting power of each user would give everyone an equal opportunity to purchase voting power for a proposal.

Code corrected:

Votes are counted at the block prior to the start of the voting period, not the proposal creation time anymore.

6.2 ActiveProposalsLimiter Doesn't Handle maxActiveProposals Decreases Correctly

In the `ActiveProposalsLimiter` contract, if the value of `maxActiveProposals` is decreased, the proposal creation limiting mechanism doesn't work as expected. If the current `activeProposals` value for a user is higher than the new `maxActiveProposals`, the user can continuously create proposals without the limit being enforced.

Code corrected:

The equality check has been replaced with a *greater or equal to* check.

6.3 Proxy Deployment Vulnerable to Front-Running

In the current implementation of the `ProxyFactory` contract, the deployment of a new proxy contract could be potentially front-run. The front-runner could deploy a proxy at the same predicted address but with different parameters.

This issue arises due to the `deployProxy` method's reliance on `salt` for proxy address prediction and the absence of the `msg.sender` or initializer payload in the address computation parameters.

One mitigation strategy would be to include the `msg.sender` or the initializer payload in the salt. This change could make the address prediction unique for every user and initializer, thus preventing potential front-run attacks. Another mitigation is to properly check that the transaction was successful when

`deployProxy()` is called, as the transaction will revert if a proxy has already been deployed at the same address.

Code corrected:

Initialization parameters and `msg.sender` are now included in the `CREATE2` salt, making it impossible for another `msg.sender` to front-run a deployment to the same address.

6.4 Space Relies Heavily on Execution Strategy for Correctness

CS-SNAPSHOT-004

The *Space* contract delegates a lot of responsibility to the execution strategies. This means that users must either trust proposal authors or check a lot of invariants on proposals they vote on. The *Space* contract could instead make some of these checks on its own, before calling the strategy.

For example, the *Space* contract should never execute a proposal that has already been executed, or one that has been cancelled. However, it doesn't check this. Additionally, it could check that a proposal that hasn't reached its `minVotingDuration` can't be executed. It would also be possible to check the `payloadHash` for equality. Similarly, one could check that a proposal hasn't been cancelled or executed before updating it.

This would reduce the possibilities for malicious proposal authors to omit necessary checks in order to trick users into trusting their proposals.

Code corrected:

Most suggested checks have been implemented:

1. The payload hash equality is checked in the *Space*.
2. In `execute()`, the finalization status has to be `Pending`.
3. In `updateProposal()`, the finalization status has to be `Pending`.

Note that `minVotingDuration` is not checked in the *Space*, as it can be overridden by some execution strategies (e.g. *EmergencyQuorumExecutionStrategy*).

6.5 TimelockExecutionStrategy Deployed as Proxy at Risk From Implementation SELFDESTRUCT

CS-SNAPSHOT-006

It is expected that *TimelockExecutionStrategy* will be deployed as a proxy contract. However, it is critical that the implementation contract shared by the proxies is disabled and cannot execute transactions. If the base implementation contract is in operation, a transaction could perform a delegate call into a contract containing the `SELFDESTRUCT` opcode, irrecoverably disabling all the proxies using that implementation.

Code corrected:

The constructor now disables the contract by transferring ownership to `address(1)`, which means no delegate call can be performed from the implementation contract.

6.6 Timestamp Resolver May Return Incorrect Block Number

CS-SNAPSHOT-007

In both `CompVotingStrategy` and `OZVotesVotingStrategy` contracts, the `TimestampResolver` is expected to return the correct block number for a given timestamp. However, it currently only guarantees the correct return if the same voting strategy is also used with a `PropositionPowerProposalValidationStrategy` at proposal creation time.

The issue arises with the method `resolveSnapshotTimestamp`, where the returned `blockNumber` doesn't necessarily correspond to the passed timestamp. `blockNumber` will only match the timestamp if `resolveSnapshotTimestamp()` is called when `block.timestamp == timestamp`. The voting power for proposals is currently queried at the proposition timestamp. The proposition timestamp (`snapshotTimestamp`) is only guaranteed to be resolved if the proposal validation strategy queries the same voting strategies as the actual voting.

Code corrected:

`TimestampResolver` has been removed from the codebase. Timestamps have been replaced with block numbers, so that no timestamp translation to block number is necessary.

6.7 `_quorumReached` Counts votesAgainst Toward the Quorum

CS-SNAPSHOT-008

The `_quorumReached` function in `SimpleQuorumExecutionStrategy` also counts `votesAgainst` toward the quorum. This creates an incentive for *Against* voters not to vote, at least until the quorum is reached. Differently from what is done here, Compound's `GovernorBravo` only counts *For* votes toward the quorum, and OpenZeppelin `Governor` counts *For* and *Abstain* votes toward the quorum.

Code corrected:

Only *For* and *Abstain* votes are counted for the quorum.

6.8 ActiveProposalsLimiter User Data Could Use a Struct

CS-SNAPSHOT-009

The ActiveProposalsLimiter contract stores the user data, which consist of a `uint32` timestamp and a `uint224` counter, in a mapping of type `uint256`. Manual packing and unpacking are therefore performed. The same thing could be achieved by using a struct, which would perform the same kind of storage packing without the need for low level operations such as bit shifts and masking.

Code corrected:

Snapshot switched to using a struct that fits in a single storage slot.

6.9 Better Storage and Struct Packing Possible

CS-SNAPSHOT-010

The order of declaration of variables in storage and the order of fields in structs are relevant to their ultimate size. Storage variables that have combined sizes less than one word can be packed into a single storage slot, which can reduce the number of storage slots used, especially if they are often read or written together anyway. The same goes for struct fields, where sequential fields can be packed into one slot if they are small enough.

This can be applied in the following places:

1. The `vp` field of the *Member* struct could be reduced to a `uint96` so the entire struct would occupy one word. 96 bits of precision is likely sufficient to measure the voting power of a user.
 2. The *Proposal* struct's fields can be reordered to reduce its size to 4 words instead of 5.
 3. The order of storage variables in the *Space* contract can be reordered to occupy fewer slots.
-

Code corrected:

1. The `vp` field in the *Member* struct has been reduced to 96 bits of precision, making it fit in a single slot.
2. The *Proposal* struct's fields have been reordered to only take 4 words.

6.10 Comp Timelock Doesn't Support Duplicate MetaTransaction Queueing

CS-SNAPSHOT-011

The `execute` function of the contract fails silently when trying to queue the same `MetaTransaction` twice within the same proposal, due to the internal workings of the Compound `Timelock` contract.

If two identical `MetaTransaction`'s are present in the payload, the two queueing attempts will only be counted as one in Compound's Timelock. This can cause seemingly valid proposals to fail when executed after the timelock delay has elapsed.

Code corrected:

The hash of each `MetaTransaction` is checked against the timelock's `queuedTransactions` so that a duplicate transaction cannot be queued.

6.11 Function Parameter Location Optimizations

CS-SNAPSHOT-012

Many functions take parameters in `memory` instead of in `calldata`. If the parameters don't need to be modified, it is more efficient not to copy them to memory unless absolutely necessary. In general, copying of entire structs should be avoided when possible.

This improvement can be made in the following places: #. All of the *Space* contract's setter functions, its `initialize` function and `_getCumulativePower`. (Note that the latter would also require an `assertNoDuplicateIndices` implementation with a `calldata` list). #. The `_assertProposalExists` takes a `Proposal` `memory` as a parameter. However, in the `execute` and `cancel` functions it is passed a `Proposal` storage variable. Thus, the entire struct will be copied into memory only to check a single field's value. #. The `bytesToAddress` function in the *CompVotingStrategy* and *OZVotingStrategy* could take a `bytes` `calldata` parameter instead of a `bytes` `memory`. #. The various `_verify` functions in the *SignatureVerifier* could take `bytes` `calldata` as a parameter.

Code corrected:

In the *Space* contract, the memory location of external functions arguments has been optimized by setting it to `calldata` as suggested.

`_verifyVoteSig()` has been modified to take `bytes` `calldata` as a parameter. The other `_verify` functions have been kept with `bytes` `memory` parameters because of stack too deep compilation errors. Since we expect voting to be the most common user action for a proposal, we are satisfied that the most visited path is optimized.

The calls to `bytesToAddress()` were replaced by a type conversion: `address(bytes20(params))`

The suggestion regarding `_assertProposalExists()` was intentionally not addressed, because it would sacrifice abstraction.

6.12 Function Visibility Can Be Restricted

CS-SNAPSHOT-013

The visibility of the following functions can be restricted in order to save gas:

1. The *Space* contract's `initialize` function can be made external.
2. The *SpaceManager* contract's `enableSpace` and `disableSpace` functions could be made external. Alternatively, an internal and external copy of the functions could exist, if inheriting contracts should have access to these functions. Additionally, the `isSpaceEnabled` function

could be made `external`, and internal uses (e.g. the `onlySpace` modifier) should directly access the `spaces` mapping to save gas.

Code corrected:

Snapshot has restricted the concerned functions' visibility to `external`.

6.13 Inconsistency in Quorum Modifiability

CS-SNAPSHOT-014

`SimpleQuorumExecutionStrategy` defines a storage variable for the quorum. It could be useful to add a setter for the quorum, as it could be useful to change it depending on the circulating voting power.

Moreover, there is an inconsistency between `SimpleQuorumExecutionStrategy` and `EmergencyQuorumStrategy` in how they handle the quorum variable. In the latter it is set as an immutable. This prevents contracts deriving from `EmergencyQuorumStrategy` from being deployed as proxies.

Code corrected:

A `setQuorum()` method has been added to `SimpleQuorumExecutionStrategy`, `OptimisticQuorumExecutionStrategy`, and `EmergencyQuorumStrategy`. `EmergencyQuorumStrategy` has been modified to use storage variables instead of immutables so that it can be initialized by cloning.

6.14 Inefficient Boolean Mappings

CS-SNAPSHOT-015

Various contracts have mappings which map to a boolean value. In Solidity, this is inefficient, because writing to such a mapping incurs an additional storage read. It is more gas-efficient to map to a type which occupies the entire size of the storage slot, e.g. a `uint256`.

Code corrected:

The following boolean mappings have been replaced with `uint256` mappings:

1. `authenticators` in `Space`.
2. `voteRegistry` in `Space`.
3. `spaces` in `SpaceManager`.
4. `usedSalts` in `SignatureVerifier`.

6.15 Multiple Voting if Voting Strategy Calls Untrusted Code

CS-SNAPSHOT-016

There is a possible reentrancy in the `vote` method of a Space. If the voting strategy calls untrusted code, `vote()` can be reentered and a user can vote several times, because the `voteRegistry` is updated only after the interactions (vote counting and incrementing) have happened.

Code corrected:

The voter's entry in the `voteRegistry` is now updated before making external calls, thus preventing any reentrant calls into the `vote` function.

6.16 No Setter for Delay in TimelockExecutionStrategy

CS-SNAPSHOT-017

No setter is defined for the `timelockDelay` storage variable in the `TimelockExecutionStrategy` contract.

Code corrected:

The `TimelockExecutionStrategy` owner can now set the timelock delay.

6.17 Potential Reentrancy in executeQueuedProposal Method

CS-SNAPSHOT-018

The `executeQueuedProposal` function, as present in both `CompTimelockCompatibleExecutionStrategy` and `TimelockExecutionStrategy`, is susceptible to potential reentrancy issues. While the method does not allow the same proposal to be executed twice by resetting the execution time to zero before any external calls, this does not protect from different proposals executing their meta-transactions in an interleaved order if one of them calls untrusted code. This could lead to unexpected outcomes, particularly when executing proposals with dependencies or conflicts.

Specification changed:

This particular behavior, which is shared with other governance systems, has been properly documented in the timelock execution strategy's *NatSpec*.

6.18 Quorum Not Included in the AvatarExecutionStrategySetUp Event Emission

CS-SNAPSHOT-020

The AvatarExecutionStrategySetUp event, emitted in the setUp function, does not include the _quorum parameter in its emission. Although it is initialized correctly and stored in the contract state, this omission could lead to potential issues with transparency and event tracking.

Code corrected:

The AvatarExecutionStrategySetUp event has been modified to include the _quorum in its data.

6.19 TimelockExecutionStrategy Not Fully ERC165 Compliant

CS-SNAPSHOT-022

The ERC165 defined supportsInterface(bytes4 interfaceId) function should return true when interfaceId == type(IERC165).interfaceId.

Code corrected:

TimelockExecutionStrategy has been modified so that supportsInterface() is ERC165 compliant.

6.20 Unnecessary Binary Search

CS-SNAPSHOT-023

The WhitelistVotingStrategy allows the Space contract to provide an ordered list of whitelisted users, so that the strategy can check that a given user is allowed to vote. It searches through this list using a binary search. However, this leads to two main problems:

1. When the admin / owner of a space updates the whitelist, this may involve writing an arbitrarily large number of storage slots, as they must reorder the entire list.
2. The gas cost of voting is higher than necessary, as every voter must run the binary search on-chain.

An alternative to this would be to provide the WhitelistVotingStrategy with an unordered list of members. When a user votes, they must provide the index of their own whitelist entry, so that the contract only needs to confirm that their address is at that index in the list, without going over any other entries. They could provide this index using the userParams field.

Code corrected:

The proposed addressing scheme has been implemented by the client.

6.21 Unused Function

CS-SNAPSHOT-024

The `_assertValidAuthenticator` function is never used.

Code corrected:

The unused function has been removed.

6.22 Various Reentrancy Possibilities During Proposal Execution

CS-SNAPSHOT-025

The `execute` function of the *Space* contract has a `nonReentrant` modifier, meaning the execution strategy can't call back into it. However, it may still call all the other functions of the *Space* contract.

If external systems rely on the state of the *Space* contract to make decisions, this may lead to problems. For example, a malicious *Space* owner could execute a strategy which cancels its own proposal, which may lead an external system to believe the proposal can no longer be executed. However, the `execute` call is already being executed. Alternatively, a proposal author could update their proposal during its execution, or even vote on it.

For example, the author could execute the proposal while it is in the *VotingAccepted* state, and in the execution provide a high number of *Against* votes so that it is no longer accepted. An external system will again not be able to know that the proposal is already being executed, despite not having enough votes to pass.

These issues stem from the fact that the `finalizationStatus` of the proposal is only set to *Executed* after the execution of the proposal. If this status change were put in storage before the call (while still providing the same proposal in memory to the execution strategy), the reentrancies could be avoided as it would be clear that the execution of the proposal has already begun.

Code corrected:

The proposal execution status is set to `Finalized` before it is executed, so reentrancies are no longer possible.

6.23 Voting Power Not Easily Estimated in User Interface

CS-SNAPSHOT-026

In the current implementation, estimating the voting power for each voting strategy of a space before casting a vote is challenging. The issue lies with the `getVotingPower()` method in the voting strategy

contracts, which is not a view function and can perform state changes due to the potential need for resolving the timestamp, in `CompVotingStrategy` and `OZVotesVotingStrategy`.

This makes it hard for the user interface to call the function for estimation purposes without sending a transaction.

Code corrected:

The `TimestampResolver` has been removed, and the mutability of the `getVotingPower()` methods has been reduced to `view`.

7 Notes

We leverage this section to highlight further findings that are not necessarily issues. The mentioned topics serve to clarify or support the report, but do not require an immediate modification inside the project. Instead, they should raise awareness in order to improve the overall understanding.

7.1 Benign Reentrancies

There are two cases in which reentrant calls could result in unexpected outcomes:

1. `ProxyFactory.deployProxy`: If the call to `proxy.call(implementation)` reentrantly calls `deployProxy` again, the *ProxyDeployed* events will be emitted in the wrong order.
2. `Space.propose`: If the call to the `ProposalValidationStrategy` results in a reentrant call (by calling untrusted code), the second call to `propose` will have a lower `proposalId`, and emit the *ProposalCreated* event first. Hence, a call to `propose` should not assume the resulting proposal will have the next available `proposalId`. Additionally, note that an on-chain actor isn't easily able to determine the ID of the proposal that was just created, as no value is returned. Instead, they would have to query the `nextProposalId` and subtract one.

7.2 Cloning and Initialization Must Happen in Same Transaction

Many contracts are expected to be deployed as proxies to an implementation contract, and then initialized through an initialization function. The contracts in question are:

1. *Space*
2. *AvatarExecutionStrategy*
3. *CompTimelockCompatibleExecutionStrategy*
4. *TimelockExecutionStrategy*

It is important that deploying proxy deployment and initialization happen in the same transaction, so that the initialization isn't vulnerable to front-running. The *ProxyFactory* contract correctly performs deployment and initialization together.

7.3 PropositionPower Strategies May Differ From Space

When the admin / owner of a *Space* changes the voting strategies of that space, it does not update the voting strategies used by a *PropositionPower*-based proposal validation strategy. Hence, they must always update the parameters of the proposal validation strategy simultaneously to keep them consistent. This additional requirement should be clearly documented.

7.4 Space Can Have Duplicate Voting Strategies

In the *Space* contract's `_getCumulativePower` function, it is asserted that a user doesn't provide the same index for a voting strategy twice. However, if the owner / admin inadvertently adds the same voting strategy twice, this index could be different for the same voting strategy, allowing the user to double count their votes.

7.5 Vanilla Contracts Should Not Be Used in Production

The various *Vanilla* contracts should not be used in production. Ideally it should be documented that these contracts are for testing purposes and not to be used for real-world systems, as using them would introduce a variety of vulnerabilities.