OpenZeppelin | security

Snapshot X Starknet Audit

October 25, 2023

Table of Contents

Summary

Type	Governance
Timeline	From 2023-09-11 To 2023-09-28
Languages	Cairo, Solidity

Total Issues	21 (16 resolved, 1 partially resolved)
Critical Severity Issues	2 (2 resolved)
High Severity Issues	3 (3 resolved)
Medium Severity Issues	1 (1 resolved)
Low Severity Issues	5 (4 resolved)
Notes & Additional Information	10 (6 resolved, 1 partially resolved)

Scope

We audited the snapshot-labs/sx-starknet repository at commit 8ce265c.

In scope were the following contracts:

```
ethereum
└── src
    ├── StarknetCommit.sol
    ├── deps.sol
    ├── execution-strategies
    │   ├── L1AvatarExecutionStrategy.sol
    │   ├── SimpleQuorumExecutionStrategy.sol
    │   └── StarknetSpaceManager.sol
    ├── interfaces
    │   ├── IExecutionStrategy.sol
    │   └── IStarknetCore.sol
    └── types.sol
starknet
└── src
    ├── authenticators
    │   ├── eth_sig.cairo
    │   ├── eth_tx.cairo
    │   ├── stark_sig.cairo
    │   └── stark_tx.cairo
    ├── execution_strategies
    │   ├── eth_relayer.cairo
    │   └── no_execution_simple_majority.cairo
    ├── external
    │   └── herodotus.cairo
    ├── factory
    │   └── factory.cairo
    ├── interfaces
    │   ├── i_execution_strategy.cairo
    │   ├── i_proposal_validation_strategy.cairo
    │   └── i_voting_strategy.cairo
    ├── lib.cairo
    ├── proposal_validation_strategies
    │   └── proposition_power.cairo
    ├── space
    │   └── space.cairo
    ├── types
    │   ├── choice.cairo
    │   ├── finalization_status.cairo
    │   ├── indexed_strategy.cairo
    │   ├── proposal.cairo
    │   ├── proposal_status.cairo
    │   ├── strategy.cairo
    │   ├── update_settings_calldata.cairo
```

```
    │   └── user_address.cairo
    ├── utils
    │   ├── bits.cairo
    │   ├── constants.cairo
    │   ├── eip712.cairo
    │   ├── endian.cairo
    │   ├── into.cairo
    │   ├── keccak.cairo
    │   ├── legacy_hash.cairo
    │   ├── math.cairo
    │   ├── merkle.cairo
    │   ├── proposition_power.cairo
    │   ├── reinitializable.cairo
    │   ├── simple_majority.cairo
    │   ├── single_slot_proof.cairo
    │   ├── stark_eip712.cairo
    │   └── struct_hash.cairo
    └── voting_strategies
        ├── erc20_votes.cairo
        ├── eth_balance_of.cairo
        └── merkle_whitelist.cairo
```

```
starknet
└── src
    └── tests
        └── mocks
            ├── simple_quorum.cairo
            └── vanilla_execution_strategy.cairo
```

In addition, we also audited the following pull requests as part of the fix review process: pull request #548, pull request #549, pull request #550, pull request #551, pull request #596, and pull request #597.

System Overview

Snapshot X is a modular on-chain voting protocol. Modules are smart contracts conforming to specific interfaces that perform authentication, proposal validation, vote calculation, and proposal execution, which allows for customization of protocol behavior.

Snapshot X Starknet is the version of the protocol deployed on Starknet. The impact, however, is not limited to Starknet because of the modular structure. For example, it is possible to have a governance token on Ethereum, create proposals and vote on Starknet, and finally execute proposals on Ethereum while providing a gasless experience for voters with Ethereum and/or Starknet wallets.

The core concepts of the protocol - proposals and spaces - are detailed below.

Proposals

Proposals represent an intention to execute transactions on-chain. Although the proposal's author chooses the details of the desired transactions (i.e., the execution strategy), the voting restrictions and timestamps are pre-configured properties of the system. Once proposed, the system manages the voting process and provides censorship-resistant execution if the proposal is accepted, with two exceptions:

- The proposal's author can change the execution strategy before the voting period begins.
- The owner can cancel any proposal before it is executed.

Proposals from different entities (e.g., different DAOs) or with different configuration requirements are separated using spaces.

Spaces

Spaces store and govern proposals related to an entity. In order to function, a space must have at least one authenticator, one proposal validation strategy, and one voting strategy. In general, a space can have multiple authenticators and voting strategies but only one proposal validation strategy.

Authenticators ensure that an author or voter indeed intended to perform a particular action. They then call the corresponding function on the space with the relevant parameters.

One of the actions is proposal creation. When a proposal is created, a proposal validation strategy checks whether the author (i.e., an address) is allowed to create a proposal. If so, the proposal is stored in the space and the pre-voting period commences, during which the proposal can be updated.

Another action involves casting a vote between the start and end timestamps. There is also a "minimum-voting-end" timestamp that execution strategies can consider when resolving the vote. The space ensures each voter (i.e., an address) can vote only once per proposal. Voting strategies determine each voter's voting power (i.e., how to weigh their vote).

In general, execution can happen at any time after proposal creation. In practice, however, it is expected that execution strategies execute a proposal only if it is accepted. The acceptance criteria are specified by the execution strategies themselves. Thus, it is the execution strategy's responsibility to verify the incoming proposal with current voting results and decide whether to execute it.

At any time before proposal execution, the owner of the space can cancel the proposal.

Spaces can be deployed by anyone using the factory. However, space settings can be updated and spaces themselves can be upgraded only by the owner of a particular space.

Authenticators

Authenticators ensure that an author or voter indeed intended to perform a particular action. There can be multiple authenticators enabled for the same space, which would provide users multiple options to authenticate their actions.

The following is an overview of four authenticators covered in this audit.

`StarkTxAuthenticator`

This authenticator uses native Starknet authentication for Starknet accounts. The calling contract must be the author or voter.

`StarkSigAuthenticator`

This authenticator accepts a signed Starknet-compatible EIP-712-like encoding of an action. This proves that the author or voter signed the action off-chain. It differs from

`EthSigAuthenticator` because the signer is a Starknet account and uses Starknet data structures.

`EthTxAuthenticator`

This authenticator consists of two components: one on each side of the Starknet bridge. The Ethereum component accepts a Poseidon hash commitment of an intended action. The hash and the calling Ethereum address are communicated to the Starknet counterpart via the Starknet bridge. Any address can reveal the action description on the Starknet side, which is confirmed to match the commitment. In this way, the authenticator allows an Ethereum address to create an authenticated action on Starknet.

`EthSigAuthenticator`

This authenticator accepts a signed EIP-712 encoding of an action to prove the author or voter signed the action off-chain with an Ethereum address private key.

Proposal Validation Strategy

The proposal validation strategy determines whether the author is allowed to make the proposal. There can be only one proposal validation strategy per space.

The following is an overview of the proposal validation strategy covered in this audit.

`PropositionPowerProposalValidationStrategy`

This strategy is configured with a required threshold of voting power and a list of allowed voting strategies, which do not necessarily match the space voting strategies. The proposal is accepted if the author had sufficient voting power across all allowed strategies immediately before the L2 block (i.e., at one second before the current timestamp).

The timestamp offset simply ensures the proposer cannot take an action within this block, like a flash loan, to increase their power. They can, however, obtain additional voting power before the block (for example, by buying tokens) and relinquish it afterwards.

Voting Strategies

Voting strategies are modules that receive an address and a timestamp, and calculate how much voting power the address had at the given timestamp. This also implicitly provides sybil resistance, since all addresses need to acquire voting power.

The following is an overview of the three voting strategies covered in this audit.

`ERC20VotesVotingStrategy`

This strategy relies on the `get_past_votes` function of an ERC20Votes token on Starknet to calculate voting power at the start of the voting period. This implicitly prevents voters from using this strategy exactly on the start timestamp, because the queried timestamp must be strictly in the past.

`EthBalanceOfVotingStrategy`

This strategy works with the Herodotus protocol to get balances of ERC-20 tokens on Ethereum to calculate voting power.

`MerkleWhitelistVotingStrategy`

This strategy works with a pre-calculated Merkle tree, the root of which is stored in the space. Each leaf of the tree contains a voter and their voting power. A plausible use case is to construct the tree using all balances of a particular token at a specific timestamp. Voters should provide their leaf and corresponding Merkle proof in order to exercise the corresponding voting power.

Proposal Execution Strategy

Proposal execution strategies are modules that are fully responsible for verifying and executing proposals. They trust the space to provide the correct and current voting state and correct proposal parameters, but must decide whether to proceed. The verification procedure may include checking voting results and checking timestamps, among other things.

The following is an overview of two proposal execution strategies covered in this audit.

`NoExecutionSimpleMajorityExecutionStrategy`

This strategy verifies that a proposal is accepted but does not actually execute it. This could be useful for a non-binding poll of voters. A proposal is considered accepted under simple majority rules: the vote has ended and there are more *for* than *against* votes.

`EthRelayerExecutionStrategy`

This strategy consists of two components: one on each side of the Starknet bridge. The Starknet part can be called by anyone and simply verifies that the voting has finished and passes the caller and the supplied proposal to the Ethereum counterpart via the Starknet bridge. On the Ethereum side it receives the message, verifies that the message originated from the space, verifies the proposal, and finally executes the proposal using the `IAvatar` interface. In this way, the Starknet space can be used to perform actions on Ethereum.

Security Model and Trust Assumptions

The space owner can perform the following actions:

- Upgrade the space (i.e., change its code by replacing the class hash).
- Update space settings such as voting delay, voting durations, metadata and DAO URIs, authenticators, proposal validation strategy, and voting strategies. Note that changing authenticators could affect active proposals.
- Cancel any proposal in the space unless it has been executed (or already canceled).
- Transfer ownership.
- Renounce ownership.

Similarly, the owner of the `L1AvatarExecutionStrategy` contract (the Ethereum component of the `EthRelayerExecutionStrategy`) can perform the following actions:

- Upgrade the whole contract.
- Update bridge settings, such as the Starknet Core contract, corresponding to the L1 side of the bridge, and the L2 contract that it receives messages from.
- Update proposal execution settings, such as the valid space contracts, the quorum required, and the target contract that will eventually be invoked.

Naturally, users of the system should trust the owner to exercise these far-ranging powers correctly and fairly.

The system assumes the Starknet timestamps progress at a predictable rate, and are approximately synchronized with the Ethereum timestamps. Since the Starknet sequencer can choose any monotonically increasing timestamps, they implicitly have the power to shorten or lengthen the voting periods.

Testing coverage recommendations

Several concerns regarding the testing of the current system were identified during the audit. Some of the critical and high issues identified cover standard behaviors that would likely have been identified by integration or end-to-end tests. In a complex modular system like the Snapshot X, we recommend testing at least the mainline cases for all modules to ensure they behave as expected. Implementing such a test suite should be a very high priority to ensure the system's robustness and reduce the risk of vulnerabilities and bugs. Additional testing coverage that leverages fuzzing and formal verification would further increase the reliability of the code.

Security Considerations for Space Owners

Here we present considerations for the space owners and module creators. They are not inherently bugs or vulnerabilities but may lead to vulnerabilities if not accounted for.

Duplicate Strategies

It is the owner's responsibility to ensure that proposal validation and voting do not contain duplicate strategies. If they do, it becomes possible to account for the same voting power multiple times, since only duplicate indices supplied by the user are prevented but there is no check for duplicate strategies.

Merkle Tree Construction

This is relevant only if the `MerkleWhitelistVotingStrategy` is used. The Merkle tree should be constructed in a way that prevents double-voting. In particular, since the strategy

does not contain information about when the voting power was measured, in the case where the voting power represents token balances, it is crucial to ensure the voting power is sampled at the same timestamp for all accounts.

Voting Strategies Cap

There is a maximum limit of 256 voting strategies for the whole lifetime of a space. This means that once 256 voting strategies have been added to the space, more cannot be added, whether or not the strategies are subsequently removed. Note that the protocol ensures that there is always at least one active voting strategy.

Race Condition in `SimpleQuorumExecutionStrategy`

The `SimpleQuorumExecutionStrategy` contract on Ethereum and Starknet may consider a vote to be accepted during the voting period. This allows the `L1AvatarExecutionStrategy` or `VanillaExecutionStrategy` to execute the corresponding proposal immediately, even though the proposal may have eventually been rejected.

The Snapshot team has indicated that this is the intended behavior. Moreover, the `EthRelayerExecutionStrategy` waits for the entire voting period before relaying the proposal to L1, so it should not be relevant in practice (unless there is a discrepancy between the L1 and L2 timestamps). Nevertheless, since some users may find this unexpected, we note it here for completeness.

Critical Severity

C-01 Votes Can Be Overcounted

When making a proposal or voting on a proposal, the total voting power across multiple strategies is combined. In both cases, the strategies are deduplicated.

This is achieved by incrementally constructing a bitmap with all the strategy indices. Each index should be checked against the bitmap before being included. However, the deduplication condition only checks index 0, so it is possible for any other strategy index to be repeated. This means that the voting power for all other strategies can be counted multiple times.

Consider ensuring the deduplication condition covers all indices. In particular, it should assert that `(bit_map & s) == 0_u256`.

Update: *Resolved in pull request #577 at commit 184afb3.*

C-02 Merkle Leaf Not Authenticated

The `MerkleWhitelistVotingStrategy` contract is configured with a Merkle root that represents a set of users and their voting power. When voting or proposing, users should provide their leaf node along with a valid Merkle proof. However, the `get_voting_power` function simply confirms that the leaf exists in the tree but does not validate that it belongs to the voter. Therefore, anyone can use the voting power of any voter.

Consider validating that the leaf matches the expected voter.

Update: *Resolved in pull request #575 at commit 87c821a.*

High Severity

H-01 Inverted Case Style

The `StarkEIP712` contract supports accounts that use both snake and camel case. However, the branch is inverted, so it attempts to use camel case on snake-case accounts and vice versa. Importantly, it checks the *ERC165* interface with the opposite case, which means the `verify_signature` function will always fail, effectively disabling the `StarkSigAuthenticator`.

Consider correcting the branch statement to use the correct case style.

Update: *Resolved in pull request #574 at commit 4afa276. The Snapshot team decided to remove support for the old account interface and only support the SNIP-6 standard.*

H-02 Votes Can Be Blocked

The `StarknetCommit` contract allows anyone to submit a hash representing their vote, which will be recorded with their L1 address in the `EthTxAuthenticator` contract. Later, they can consume the commitment, provided the hash matches a valid transaction (i.e., proposal, vote, or proposal update) from the original sender.

However, an attacker can submit a hash corresponding to another user's valid transaction. This will prevent the victim from making the same commitment to execute the transaction. Fortunately, since metadata_uri is a free parameter, they could commit to an equivalent transaction with a different `metadata_uri` value, and use a private relayer to ensure they cannot be front-run.

Consider including the L1 message sender as part of the hash commitment to ensure uniqueness. Alternatively, consider using a two-dimensional storage mapping to allow multiple users to commit to the same hash.

Update: *Resolved in pull request #593 at commit cddb750.*

H-03 Incorrect Slot Key

The `get_mapping_slot_key` function accepts a key and slot index in that order, but the `get_storage_slot` function passes the parameters in reverse order. This means it will calculate the wrong slot key and the storage proof will fail. Consequently, the `EthBalanceOfVotingStrategy` cannot retrieve the voter's balance, making the strategy unusable.

Consider passing the supplied arguments in the same order as the function signature.

Update: *Resolved in pull request #580 at commit ed6ef10.*

Medium Severity

M-01 Space Deployment Vulnerable To Front-Running

The `deploy` function of the factory is meant to deploy new spaces. However, it can be front-run with the same class hash and salt but different initialization calldata. As a result, the attacker may set themselves as the owner of the victim's newly deployed space. The victim may miss or ignore the error messages, think that the space was deployed correctly and then proceed to use it.

Consider adding the caller address to the salt in order to bind the contract address to the deployer.

Update: *Resolved in pull request #581 at commit c2931b8.*

Low Severity

L-01 Unused Strategy Slot

The number of voting strategies per space contract should be limited to 256, so a single 256 bitmap can represent them all. This means the last available index should be 255. However, it is actually limited to strictly less than 255. As an example, if the next index is 255 and there is

one voting strategy to add, this should be valid, but the assertion will fail since both sides of the inequality are 255.

Consider updating the range check to use an inclusive bound (i.e., `<=`).

Update: *Resolved in pull request #576 at commit 1f4ece5.*

L-02 No Gap Variable

The `StarknetSpaceManager` and `SimpleQuorumExecutionStrategy` contracts are upgradeable and ancestor contracts.

Whenever upgradeable contracts are inherited, consider including a storage gap to safely support local changes to the storage layout.

Update: *Resolved in pull request #591 at commit 78705bf.*

L-03 Functional Implementation Contract

The `L1AvatarExecutionStrategy` is intended to be accessed with a proxy pattern. The implementation should be disabled as much as possible to limit the attack surface, but it is initialized with user-supplied parameters.

To further reduce its functionality, consider ensuring there is no valid owner, target or space.

Update*: Acknowledged, not resolved. The Snapshot team stated:*

> *We would like to maintain the possibility of using the strategy without a proxy and therefore would prefer not to enforce that the implementation gets disabled. We ensure any implementation used with proxies is disabled via our deployment process using a `CREATE2` factory.*

L-04 Risky Upgrade Pattern

The space is upgraded by replacing the class hash and then calling the new initializer. However, the new initializer has to be designed specifically as an upgrade function, with reference to the current contract state. In particular, if the new initializer is similar to the current one, it would reset the next proposal id. It would also require the upgrade to include new voting strategies and authenticators. On the other hand, if it is designed as an upgrader and does not

affect these parameters, it would not function as a complete initializer if the new contract was deployed independently.

Instead of calling the new initializer, consider calling a separate single-use `upgrade` function that accounts for the current contract state. Alternatively, consider using the versioned initializer pattern.

Update: *Resolved in pull request #584 at commit badb1f4 and pull request #598 at commit 15b17a6.*

L-05 Infinite Loop

The `into_le_u64_array` breaks the loop after processing the last element but does not break when the loop is empty. This means that if the input array is empty, the function enters an infinite loop.

Consider raising an error in this case.

Update: *Resolved in pull request #579 at commit fd60237.*

Notes & Additional Information

N-01 Inconsistent Storage Pattern

The codebase mostly does not declare storage variables that are not specifically required, even if they are implicitly part of the storage. For example, the `Space` storage struct does not include an owner, even though it is implicitly referenced. However, the `EthSigAuthenticator` contract does declare a `_domain_hash` variable that is only implicitly referenced within the contract.

Consider choosing one of the patterns and applying it consistently throughout the codebase.

Update: *Resolved in pull request #583 at commit 3ef34e1.*

N-02 Code Clarity Suggestions

The following are some suggestions to improve the clarity of the codebase:

- When reading `felt252` arrays from storage, the pointer offset is increased by the `felt252` size, but the exit condition does not account for size. This behaves correctly because the size is actually 1, but we believe it would still be clearer to include it consistently.
- This hardcoded constant could use BoundedU8::max for readability.
- This hardcoded address could use `zero` for readability.
- This addition is unnecessary because it is equivalent to `with_prefix = u256 {low: input, high: prefix}`.
- The `MerkleWhitelistVotingStrategy` contract seems to unnecessarily split the `user_params` so they can be deserialized independently, instead of deserializing them together.
- The `uint256_into_le_u64s` function could return the components in a systematic order (i.e., most-significant 64-bit value to least-significant 64-bit value).
- This error message could be changed to `Already finalized` for consistency with the rest of the codebase.
- These two checks ([1] and [2]) can leverage the `is_non_zero` function.

Update: *Resolved in pull request #588 at commit 4a01e00.*

N-03 Unused Proposal Status

The `NoExecutionSimpleMajorityExecutionStrategy` contract handles the `VotingPeriodAccepted` status during execution even though it can never be in that state.

To improve the consistency of the codebase, consider removing this option.

Update: *Resolved in pull request #586 at commit 276a810.*

N-04 Typographical Errors

We identified the following typographical errors:

- "agasint" should be "against".
- "Reentrancy" is misspelled here and here.

Consider correcting these typographical errors to improve the readability of the codebase.

Update: *Resolved in pull request #578 at commit afc6644.*

N-05 Voting Strategy Limitations

The `Space` contract could be more robust in its management of voting strategies. In particular, there could be a mechanism to re-enable removed strategies. This would allow the `add_voting_strategies` function to prevent duplicates.

Update*: Acknowledged, not resolved. The Snapshot team stated:*

> *Decided against a change here as the benefit did not seem worth the architectural change.*

N-06 Incomplete Encapsulation

For efficiency reasons, the `StarknetSpaceManager` uses `uint256` values to internally represent booleans. However, the `uint256` is also returned from external queries, which requires the caller to know the convention.

In the interest of predictability, consider returning a `boolean` when queried externally.

Update*: Acknowledged, not resolved. The Snapshot team stated:*

> *Decided against a change here as the benefit did not seem worth the architectural change.*

N-07 Duplicate Code

The default contract address is implemented in `user_address.cairo` and `proposal.cairo`.

Consider removing the redundant version.

Update: *Resolved in pull request #582 at commit 1872fed.*

N-08 Misleading Comments

The following comments are misleading or imprecise:

- This comment says "is not pending" when it should say "is pending".
- This error message does not match the condition it is checking, since the "caller" is the authenticator contract.

Consider updating these comments accordingly.

Update: *Resolved in pull request #573 at commit b821347.*

N-09 Unindexed Events

The `StarknetSpaceManager` events are not indexed.

Consider indexing the `space` parameter to facilitate off-chain processing.

Update*: Acknowledged, not resolved. The Snapshot team stated:*

> *Decided against this as it increases gas costs and our off-chain indexer does not require it.*

N-10 Naming Suggestions

The following function variables could benefit from more descriptive naming:

- The proposal validation strategy validates that the author can make a proposal, rather than validating anything about the proposal itself, and could be renamed accordingly.
- The `update_proposal` function could be renamed to `update_proposal_execution_strategy`, since most of the proposal parameters cannot be changed.
- The authenticator contracts could rename `target` to `space` in all functions to better distinguish it from the proposal target.
- The `l1_account_address` variable does not refer to an account, and could be renamed to `l1_token_address` or `l1_contract_address`.
- The `reinitialize` function merely prepares the contract for a future reinitialization, and could be renamed to `reset` or `allow_reinitialization`.

- The *Eth* prefix in the `EthBalanceOfVotingStrategy` contract refers to *Ethereum*, but could be mistaken for *ETH* (the asset). It could be renamed to `L1BalanceOfVotingStrategy`.

Update: *Partially resolved in pull request #590 at commit cf4f365. The Snapshot team stated:*

> *Decided against updating the `update_proposal` and `validate` functions to keep parity with SX-EVM and not be too verbose.*

Conclusion

This audit was conducted over the course of 2.5 weeks. Overall, the codebase was of good quality and the Snapshot team was very responsive providing us with the necessary details to understand the overall system in scope of the audit.