Security Review Report
NM-0096 NounsDAO Prop-House

NETHERMIND
SECURITY

(Aug 28, 2023)

Contents

1 Executive Summary **2**

2 Audited Files **3**
2.1 Solidity Contracts . . . 3
2.2 Cairo Contracts . . . 4

3 Summary of Issues **5**
3.1 L1 Issues . . . 5
3.2 L2 Issues . . . 5
3.3 Cross-Layer Issues . . . 5

4 Protocol Overview **6**
4.1 Ethereum layer overview . . . 6
4.2 Starknet layer overview . . . 7

5 Risk Rating Methodology **8**

6 Findings **9**
6.1 [High] Bit-shift error in `StorageSlotIntoU256` causes all L1 storage queries revert . . . 9
6.2 [High] Infinite round voting can be manipulated. . . . 10
6.3 [High] Timestamp to block number entries can be manipulated . . . 11
6.4 [Medium] An infinite number of proposals can be created . . . 12
6.5 [Medium] Highly voted proposals can be cancelled to increase weighting of remaining unspent votes . . . 13
6.6 [Medium] Incremental Merkle subtree is not updated upon proposal approval . . . 14
6.7 [Medium] Spending voting power with multiple voting parameters are miscalculated . . . 15
6.8 [Medium] The `startTimestamp` in the L2 message payload is not modified when creating an infinite round . . . 16
6.9 [Low] Assets cannot be reclaimed in infinite round while state is in `FinalizationPending` . . . 17
6.10 [Low] Users can't re-vote on updated proposals . . . 18
6.11 [Info] Array storage limit not checked in implementation . . . 19
6.12 [Info] Assets can be deposited to round during `FINALIZED`, `FINALIZED_PENDING` and `CANCELLED` state . . . 19
6.13 [Info] Incorrect comment related to function selector . . . 19
6.14 [Info] Protocol does not support message cancellation . . . 20
6.15 [Info] `SpanStorageAccess::size_internal` can infinitely recurse . . . 20
6.16 [Best Practices] Avoid using `downcast` directly . . . 20
6.17 [Best Practices] Ethereum addresses stored on Starknet should be type `EthAddress` . . . 21
6.18 [Best Practices] Function `get_power` can have a `view` visibility . . . 21
6.19 [Best Practices] Function `get_slot_key(...)` cannot query a mapping with a key higher than `felt252` size . . . 22
6.20 [Best Practices] Function `get_slot_value(...)` reverts if Ethereum storage slot has value `0` . . . 22
6.21 [Best Practices] Public functions can be external . . . 23
6.22 [Best Practices] Unnecessary mutable variables/arguments . . . 23
6.23 [Best Practices] Unnecessary setting of `state` in round contracts . . . 24
6.24 [Best Practices] Unnecessary snap and desnap operations . . . 24
6.25 [Best Practices] Unnecessary variable `i` in `get_cumulative_governance_power(...)` . . . 25

7 Documentation Evaluation **26**

8 Test Suite Evaluation **27**
8.1 Contracts Compilation . . . 27
8.1.1 Solidity . . . 27
8.1.2 Cairo . . . 27
8.2 Tests Output . . . 27
8.2.1 Solidity . . . 27
8.2.2 Cairo . . . 29
8.2.3 Crosschain . . . 29
8.3 Tests Summary . . . 29

9 About Nethermind **30**

1 Executive Summary

This document presents the security review performed by Nethermind on the Prop-House protocol by Nouns DAO. Prop-House is a novel approach to community-based funding, where each community can have a "house" from which many funding "rounds" can be created. During a funding round, users can submit proposals which are then voted by the community. There are two types of rounds currently supported; timed rounds (where winners are calculated after some period of time) and infinite rounds (where winners are calculated at an arbitrary point in time decided by the round creator). Voting power can be calculated in different ways through voting strategies, which at the time of this audit included token amounts held and pre-determined weighting via a merkle tree. There can be multiple winners for a given round, and it is possible for assets to be distributed to winners. Prop-House leverages the cheaper execution cost of Starknet for all voting logic and winner calculations, where the final winner results are sent to Ethereum.

The audit was performed using (a) manual analysis of the codebase, (b) automated analysis tools, (c) simulation of the smart contract, and (d) creation of test cases. **Along this document, we report** 25 points of attention, where 3 are classified as `High`, 5 are classified as `Medium`, 2 are classified as `Low`, 5 are classified as `Info` and 10 are classified as `Best Practices`. The issues are summarized in Fig. 1.

This document is organized as follows. Section 2 presents the files in the scope of this audit. Section 3 summarizes the issues. Section 4 presents the system overview. Section 5 discusses the risk rating methodology adopted for this audit. Section 6 details the issues. Section 7 discusses the documentation provided by the client for this audit. Section 8 presents the compilation, tests, and automated tests. Section 9 concludes the document.

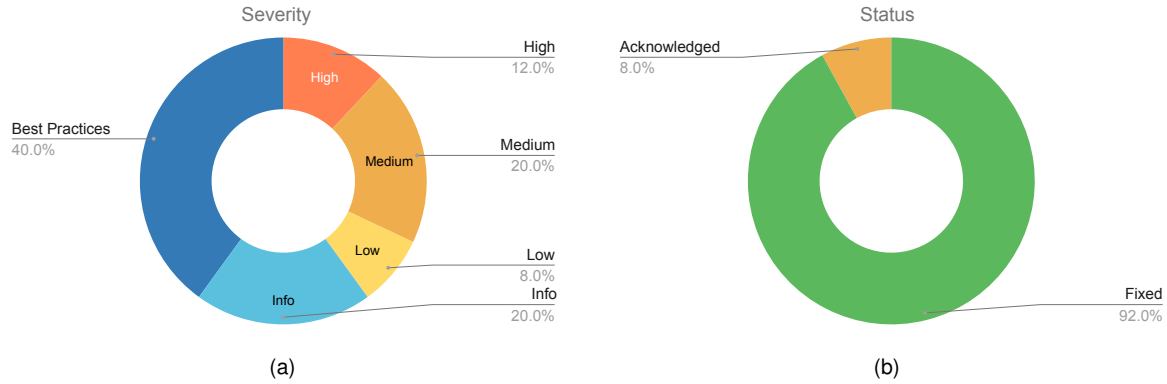

Fig. 1: Distribution of issues: Critical (0), **High** (3), **Medium** (5), **Low** (2), **Undetermined** (0), **Informational** (5), **Best Practices** (10). **Distribution of status: Fixed** (23), **Acknowledged** (2), **Mitigated** (0), **Unresolved** (0), **Partially Fixed** (0)

Summary of the Audit

Audit Type	Security Review
Initial Report	Jul. 26, 2023
Response from Client	-
Final Report	-
Methods	Manual Review, Automated Analysis
Repository	https://github.com/Prop-House/prop-house-monorepo/
Commit Hash (Audit)	c3f5ebfe914fdf498e250b46e7566a4cea960929
Commit Hash (Fix Review)	6c67d0c032b5833d7cfd885b43b399ee0560fcd4
Documentation	README.md
Documentation Assessment	High
Test Suite Assessment	Solidity: High, Cairo: Low

2 Audited Files

2.1 Solidity Contracts

	Contract	LoC	Comments	Ratio	Blank	Total
1	ethereum/Constants.sol	23	19	82.6%	14	56
2	ethereum/Manager.sol	34	22	64.7%	14	70
3	ethereum/CreatorPassIssuer.sol	68	32	47.1%	18	118
4	ethereum/Messenger.sol	47	23	48.9%	10	80
5	ethereum/PropHouse.sol	175	68	38.9%	45	288
6	ethereum/renderers/AssetMetadataRenderer.sol	46	11	23.9%	8	65
7	ethereum/renderers/TimedRoundMetadataRenderer.sol	15	3	20.0%	5	23
8	ethereum/renderers/CommunityHouseMetadataRenderer.sol	15	3	20.0%	5	23
9	ethereum/renderers/encoders/MetadataEncoder.sol	26	9	34.6%	4	39
20	ethereum/rounds/InfiniteRound.sol	210	113	53.8%	56	379
21	ethereum/rounds/TimedRound.sol	203	147	72.4%	50	400
22	ethereum/rounds/base/AssetRound.sol	65	41	63.1%	17	123
23	ethereum/rounds/base/Round.sol	90	43	47.8%	25	158
24	ethereum/lib/utils/ERC165.sol	7	2	28.6%	2	11
25	ethereum/lib/utils/Uint256.sol	15	9	60.0%	4	28
26	ethereum/lib/utils/TokenReceiver.sol	27	3	11.1%	4	34
27	ethereum/lib/utils/Sort.sol	27	5	18.5%	2	34
28	ethereum/lib/utils/Initializable.sol	41	12	29.3%	20	73
29	ethereum/lib/utils/AssetHelper.sol	38	10	26.3%	7	55
30	ethereum/lib/utils/DepositReceiver.sol	7	1	14.3%	2	10
31	ethereum/lib/utils/TokenHolder.sol	12	4	33.3%	2	18
32	ethereum/lib/utils/MerkleProof.sol	20	11	55.0%	1	32
33	ethereum/lib/utils/AssetController.sol	102	45	44.1%	20	167
34	ethereum/lib/utils/Ownable.sol	46	23	50.0%	20	89
35	ethereum/lib/utils/Address.sol	29	11	37.9%	8	48
36	ethereum/lib/utils/IncrementalMerkleProof.sol	29	4	13.8%	5	38
37	ethereum/lib/utils/ImmutableStrings.sol	21	6	28.6%	4	31
38	ethereum/lib/types/Common.sol	21	5	23.8%	4	30
39	ethereum/lib/token/ERC721.sol	121	70	57.9%	61	252
40	ethereum/lib/token/ERC1155Supply.sol	40	5	12.5%	8	53
41	ethereum/lib/token/ERC1155.sol	175	26	14.9%	48	249
42	ethereum/houses/CommunityHouse.sol	85	47	55.3%	23	155
43	ethereum/interfaces/ITimedRound.sol	51	33	64.7%	22	106
44	ethereum/interfaces/IHouse.sol	11	14	127.3%	8	33
45	ethereum/interfaces/IAssetRound.sol	9	13	144.4%	5	27
46	ethereum/interfaces/IOwnable.sol	14	22	157.1%	11	47
47	ethereum/interfaces/IRound.sol	11	11	100.0%	8	30
48	ethereum/interfaces/IStarknetCore.sol	21	18	85.7%	4	43
49	ethereum/interfaces/IMetadataEncoder.sol	9	8	88.9%	2	19
50	ethereum/interfaces/ITokenMetadataRenderer.sol	4	3	75.0%	1	8
51	ethereum/interfaces/IERC721.sol	22	51	231.8%	19	92
52	ethereum/interfaces/IMessenger.sol	23	17	73.9%	6	46
53	ethereum/interfaces/IERC1155.sol	31	17	54.8%	13	61
54	ethereum/interfaces/IERC165.sol	4	8	200.0%	1	13
55	ethereum/interfaces/IPropHouse.sol	34	33	97.1%	15	82
56	ethereum/interfaces/IDepositReceiver.sol	6	4	66.7%	3	13
57	ethereum/interfaces/IManager.sol	9	19	211.1%	6	34
58	ethereum/interfaces/IInitializable.sol	8	8	100.0%	5	21
59	ethereum/interfaces/ICreatorPassIssuer.sol	12	26	216.7%	9	47
60	ethereum/interfaces/IInfiniteRound.sol	49	32	65.3%	21	102
	Total	**2476**	**1208**	**48.8%**	**743**	**4427**

2.2 Cairo Contracts

	Contract	LoC	Comments	Ratio	Blank	Total
1	starknet/src/lib.cairo	4	0	0.0%	0	4
2	starknet/src/common.cairo	5	0	0.0%	0	5
3	starknet/src/factories.cairo	1	0	0.0%	0	1
5	starknet/src/rounds.cairo	2	0	0.0%	0	2
6	starknet/src/factories/ethereum.cairo	77	30	39.0%	21	128
7	starknet/src/rounds/timed.cairo	4	0	0.0%	0	4
8	starknet/src/rounds/infinite.cairo	4	0	0.0%	0	4
9	starknet/src/rounds/timed/constants.cairo	14	17	121.4%	9	40
10	starknet/src/rounds/timed/round.cairo	513	137	26.7%	107	757
11	starknet/src/rounds/timed/config.cairo	233	31	13.3%	21	285
12	starknet/src/rounds/timed/auth.cairo	2	0	0.0%	0	2
13	starknet/src/rounds/timed/auth/ethereum_sig.cairo	310	85	27.4%	40	435
14	starknet/src/rounds/timed/auth/ethereum_tx.cairo	169	35	20.7%	20	224
15	starknet/src/rounds/infinite/constants.cairo	15	18	120.0%	10	43
16	starknet/src/rounds/infinite/round.cairo	419	101	24.1%	87	607
17	starknet/src/rounds/infinite/config.cairo	283	31	11.0%	24	338
18	starknet/src/rounds/infinite/auth.cairo	2	0	0.0%	0	2
19	starknet/src/rounds/infinite/auth/ethereum_sig.cairo	337	89	26.4%	41	467
20	starknet/src/rounds/infinite/auth/ethereum_tx.cairo	188	37	19.7%	20	245
24	starknet/src/common/power.cairo	3	0	0.0%	0	3
25	starknet/src/common/execution.cairo	1	0	0.0%	0	1
26	starknet/src/common/registry.cairo	3	0	0.0%	0	3
27	starknet/src/common/utils.cairo	12	0	0.0%	0	12
28	starknet/src/common/libraries.cairo	4	0	0.0%	0	4
29	starknet/src/common/utils/traits.cairo	24	4	16.7%	4	32
30	starknet/src/common/utils/math.cairo	7	3	42.9%	0	10
31	starknet/src/common/utils/constants.cairo	33	11	33.3%	8	52
32	starknet/src/common/utils/array.cairo	123	14	11.4%	16	153
33	starknet/src/common/utils/integer.cairo	49	0	0.0%	6	55
34	starknet/src/common/utils/merkle.cairo	209	62	29.7%	34	305
35	starknet/src/common/utils/serde.cairo	16	0	0.0%	1	17
36	starknet/src/common/utils/contract.cairo	13	8	61.5%	3	24
37	starknet/src/common/utils/storage.cairo	70	14	20.0%	7	91
38	starknet/src/common/utils/signature.cairo	58	4	6.9%	8	70
39	starknet/src/common/utils/hash.cairo	14	4	28.6%	2	20
40	starknet/src/common/utils/bool.cairo	21	0	0.0%	2	23
41	starknet/src/common/libraries/ownable.cairo	36	11	30.6%	9	56
42	starknet/src/common/libraries/round.cairo	161	25	15.5%	23	209
43	starknet/src/common/libraries/commit_receiver.cairo	24	16	66.7%	7	47
44	starknet/src/common/libraries/single_slot_proof.cairo	93	17	18.3%	18	128
45	starknet/src/common/execution/ethereum.cairo	36	10	27.8%	10	56
46	starknet/src/common/power/allowlist.cairo	40	18	45.0%	12	70
47	starknet/src/common/power/vanilla.cairo	16	7	43.8%	2	25
48	starknet/src/common/power/ethereum_balance_of.cairo	36	10	27.8%	7	53
49	starknet/src/common/registry/round_dependency.cairo	77	55	71.4%	27	159
50	starknet/src/common/registry/ethereum_block.cairo	38	20	52.6%	8	66
51	starknet/src/common/registry/strategy.cairo	102	20	19.6%	17	139
	Total	**4231**	**977**	**23.1%**	**684**	**5892**

3 Summary of Issues

3.1 L1 Issues

Finding	Severity	Update
Assets can be deposited to round during `FINALIZED` , `FINALIZED_PENDING` , and `CANCELLED` state	Info	Acknowledged
Public functions can be external	Best Practices	Fixed
Unnecessary setting of `state` in round contracts	Best Practices	Fixed

3.2 L2 Issues

Finding	Severity	Update
Bit-shift error in `StorageSlotIntoU256` causes all L1 storage queries revert	High	Fixed
Infinite round voting can be manipulated.	High	Fixed
Timestamp to block number entries can be manipulated	High	Fixed
An infinite number of proposals can be created	Medium	Fixed
Highly voted proposals can be canceled to increase weighting of remaining unspent votes	Medium	Fixed
Incremental Merkle subtree is not updated upon proposal approval	Medium	Fixed
Spending voting power with multiple voting parameters are miscalculated	Medium	Fixed
Users can't re-vote on updated proposals	Low	Fixed
Incorrect comment related to function selector	Info	Fixed
`SpanStorageAccess::size_internal` can infinitely recurse	Info	Fixed
Array storage limit not checked in implementation	Info	Fixed
Avoid using `downcast` directly	Best Practices	Fixed
Ethereum addresses stored on Starknet should be type `EthAddress`	Best Practices	Fixed
Function `get_power` can have a `view` visibility	Best Practices	Fixed
Function `get_slot_key(...)` cannot query a mapping with a key higher than `felt252` size	Best Practices	Fixed
Unnecessary mutable variables/arguments	Best Practices	Fixed
Unnecessary snap and desnap operations	Best Practices	Fixed
Unnecessary variable `i` in `get_cumulative_governance_power(...)`	Best Practices	Fixed

3.3 Cross-Layer Issues

Finding	Severity	Update
The `startTimestamp` in the L2 message payload is not modified when creating an infinite round	Medium	Fixed
Assets cannot be reclaimed in infinite round while state is in `FinalizationPending`	Low	Fixed
Protocol does not support message cancellation	Info	Fixed
Function `get_slot_value(...)` reverts if Ethereum storage slot has value `0`	Best Practices	Fixed

4 Protocol Overview

The Prop-House protocol can be split into the Ethereum and Starknet layers. A user will create a round on the Ethereum layer, where this round data is then sent to Starknet. Prop-House utilizes the cheaper execution cost of Starknet to make the voting process and winner calculations more affordable. Users will then create and vote on proposals on Starknet before winners are chosen, and a Merkle root containing winner information is returned back to Ethereum. Winners can then prove their existence in the Merkle tree on Ethereum and claim rewards. Below, we show the contract interaction graphs for each layer and describe each contract and its purpose.

4.1 Ethereum layer overview

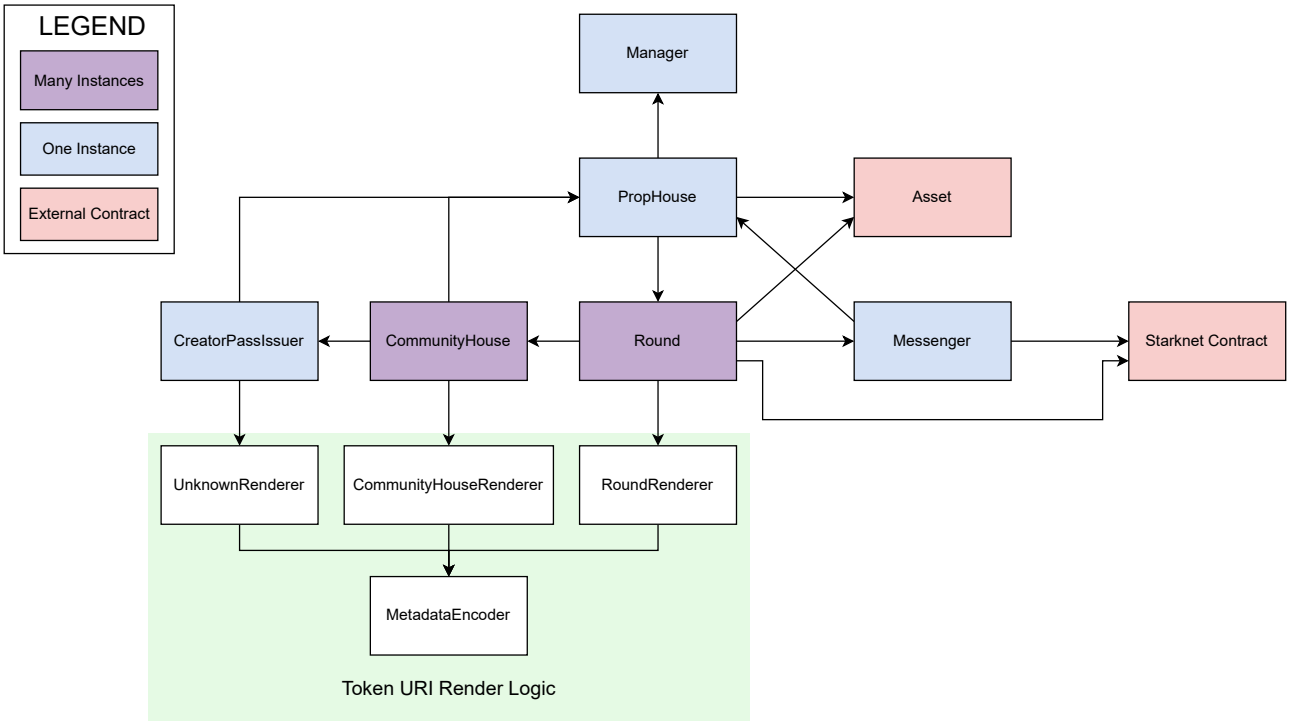

Fig. 2: Prop-House - Structural Diagram of Ethereum Contracts

PropHouse: Main entrypoint which users interact with. Allows anyone to deploy a new house, create and/or deposit to a round.

Round: Can be a timed round or infinite round. Once the round is created, users can submit proposals on Starknet. When a round is finalized, a Merkle tree of the winners is passed from Starknet back to this contract, where winners can claim assets.

Manager: Tracks valid house and round implementations, which can be cloned by the PropHouse contract.

CommunityHouse: A house is owned by one address, and any number of addresses can be assigned a creator pass. Users with creator passes can create rounds for that given house.

CreatorPassIssuer: Tracks the assigned creator passes for all houses. Can add or remove passes from an address, also can query if a particular address holds a pass.

Messenger: Wrapper contract for making calls to the Starknet contract for message sending and cancellation. Message consumption is handled directly without this contract.

Asset: An external asset used for round asset deposits. Can be ERC20, ERC721, ERC1155 or Ether.

Render logic contracts: These contracts are exclusively used for tokens with a `uri()` function, and exist to help render the token URI.

Starknet Contract: The Starknet contract on Ethereum which is used to communicate with the Starknet chain.

4.2 Starknet layer overview

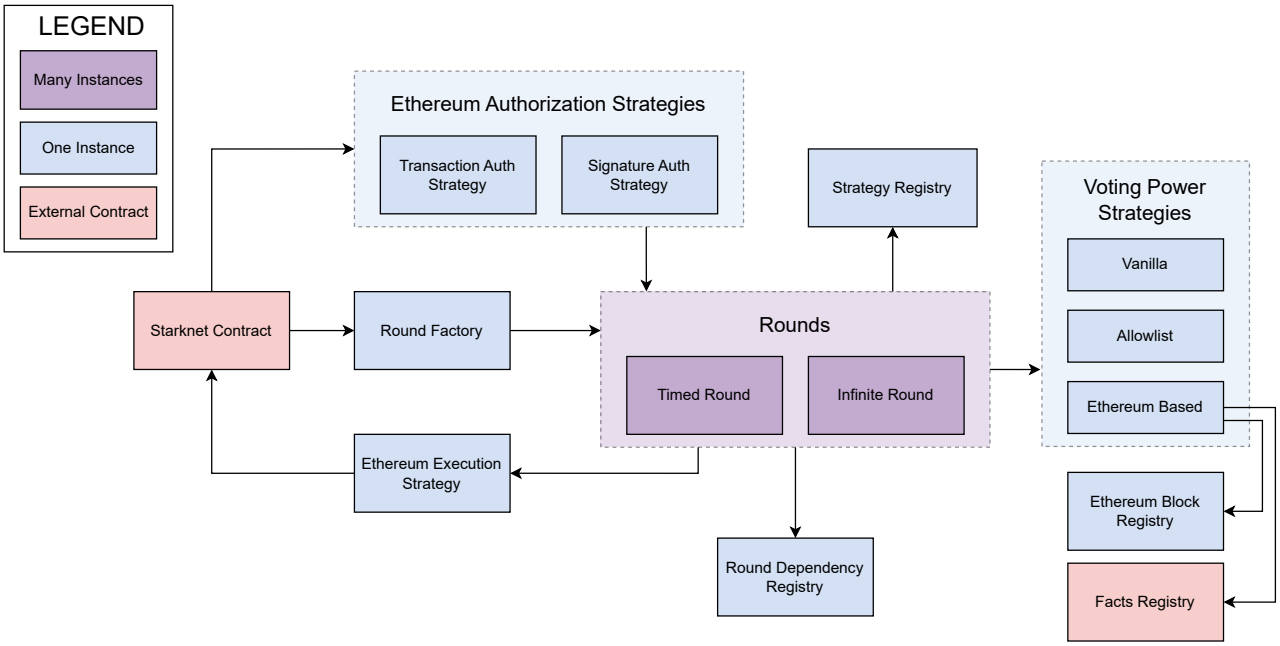

Fig. 3: Prop-House - Structural Diagram of Starknet Contracts

Timed Round: This round type can be broken down into three separate stages. First is the proposal submission stage, where users can submit proposals to the round, but no votes can be cast yet. Second, is the voting phase, where votes can be made on the proposals. After these two stages, there is the finalization stage, where the proposals with the most votes are compiled into a Merkle tree that is sent to the matching round contract on the Ethereum layer.

Infinite Round: This round type doesn't have a specified end. Instead, only the deployer of the round is able to start the finalization process. While the round has not yet been finalized, users can vote in favor or against proposals. If a certain threshold of "for" or "against" votes is met, then a proposal can be considered accepted or rejected. In the case of an accepted proposal, it is added to the Merkle tree that will eventually be sent to the matching round contract on the Ethereum layer.

Transaction Auth Strategy: Given a commit hash containing relevant information for a voting/proposal call to a round which has been sent from the Ethereum layer, make a call to the Starknet round contract with the same arguments used to derive the hash. This can be executed by any account on Starknet and allows for Ethereum users to vote on Starknet without the need for their own Starknet wallet.

Signature Auth Strategy: Given a message signed by an EOA on Ethereum, if the signature can be verified to be correct, then execute a call to interact with the round on behalf of the user who signed the message.

Round Factory: Receives messages from Ethereum to create new rounds and finalize them. New rounds are created using the `REGISTER_ROUND` selector and rounds are finalized or canceled using the `ROUTE_CALL` selector.

Ethereum Execution Strategy: This smart contract should be paired with a round factory. It will use information from the round factory to relay information from any round to the equivalent round on the Ethereum layer. Starknet to Ethereum communication would happen when a round is finalized or when winners are being processed in an infinite round.

Strategy Registry: Acts as universal storage for all defined governance strategies.

Round Dependency Registry: This contract stores immutable dependencies such as authentication and execution strategies for rounds. It uses a combination of the origin chain ID and round type as the dependency key, which allows the Starknet rounds to service many origin chains. Rounds can only read dependencies from this contract if they have been permanently locked.

Vanilla: This is for testing only, all users have a power of 1.

Allowlist: A predefined list of accounts with specified voting weights.

Ethereum Based: Uses Herodotus V1 to determine governance power using ERC20 or ERC721 balance on the Ethereum layer.

Ethereum Block Registry: This contract maps timestamps to Ethereum block numbers by reading from the Herodotus V1 headers store.

Facts Registry: Used to query Ethereum storage from Starknet.

Starknet: This is not a real contract, but exists on the diagram to represent information that travels from Starknet to Ethereum.

5 Risk Rating Methodology

The risk rating methodology used by Nethermind follows the principles established by the OWASP Foundation. The severity of each finding is determined by two factors: **Likelihood** and **Impact**.

Likelihood measures how likely an attacker will uncover and exploit the finding. This factor will be one of the following values:

a) **High**: The issue is trivial to exploit and has no specific conditions that need to be met;

b) **Medium**: The issue is moderately complex and may have some conditions that need to be met;

c) **Low**: The issue is very complex and requires very specific conditions to be met.

When defining the likelihood of a finding, other factors are also considered. These can include but are not limited to Motive, opportunity, exploit accessibility, ease of discovery, and ease of exploit.

Impact is a measure of the damage that may be caused if an attacker exploited the finding. This factor will be one of the following values:

a) **High**: The issue can cause significant damage, such as loss of funds or the protocol entering an unrecoverable state;

b) **Medium**: The issue can cause moderate damage, such as impacts that only affect a small group of users or only a particular part of the protocol;

c) **Low**: The issue can cause little to no damage, such as bugs that are easily recoverable or cause unexpected interactions that cause minor inconveniences.

When defining the impact of a finding other factors are also considered. These can include but are not limited to Data/state integrity, loss of availability, financial loss, and reputation damage. After defining the likelihood and impact of an issue, the severity can be determined according to the table below.

		Severity Risk		
Impact	**High**	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Info/Best Practices	Low	Medium
	Undetermined	Undetermined	Undetermined	Undetermined
		Low	**Medium**	**High**
		Likelihood		

To address issues that do not fit a High/Medium/Low severity, Nethermind also uses three more finding severities: **Informational**, **Best Practices**, and **Undetermined**.

a) **Informational** findings do not pose any risk to the application, but they carry some information that the audit team intends to pass to the client formally;

b) **Best Practice** findings are used when some piece of code does not conform with smart contract development best practices;

c) **Undetermined** findings are used when we cannot predict the impact or likelihood of the issue.

6 Findings

6.1 [High] Bit-shift error in `StorageSlotIntoU256` causes all L1 storage queries revert

File(s): single_slot_proof.cairo

Description: The struct type `StorageSlot` is used in the function `get_slot_value(...)` to represent an Ethereum storage slot. It contains four `felt252` values representing 64 bits of data each. The struct implements the `TryInto` trait allowing the struct representation of the storage slot to be converted into a `u256` type instead. The function is shown below:

```
impl StorageSlotIntoU256 of Into<StorageSlot, u256> {
    fn into(self: StorageSlot) -> u256 {
        let word_1_shifted = self.word_1 * UINT64_MAX_FELT;
        let word_3_shifted = self.word_3 * UINT64_MAX_FELT;
        let low = (word_3_shifted + self.word_4).try_into().unwrap();
        let high = (word_1_shifted + self.word_2).try_into().unwrap();

        u256 { low, high }
    }
}
```

When converting from `StorageSlot` to `u256` the first and third entries in the struct are left-shifted 64 bits by multiplying with `UINT64_MAX_FELT`, which has the value `0xffffffffffffffff`, or `2**64-1`. This does not result in the correct shift output however. The first and third entries should be multiplied by `2**64` instead of `2**64-1`. Consider the following example below:

```
// Incorrect left-shift by multiplying with 2^64-1
0xaa * 0x0ffffffffffffffff = 0xa9fffffffffffff56

// Correct left-shift by multiplying with 2^64
0xaa * 0x10000000000000000 = 0xaa0000000000000000
```

This causes all conversions from `StorageSlot` to `u256` to result in an incorrect value, which will cause the following assert to fail on every call, preventing any transactions that involve L1 storage queries from executing.

```
// Ensure the slot proof is for the correct slot
let valid_slot = get_slot_key(slot_index, user);
assert(slot.into() == valid_slot, 'SSP: Invalid slot');
```

Recommendation(s): Change the shift calculation to use `2**64` instead of `2**64-1`.

Status: Fixed

Update from the client: Fixed in e231305. The shift calculation now uses `2**64`.

6.2 [High] Infinite round voting can be manipulated.

File(s): infinite/round.cairo

Description: For infinite rounds in the function `_cast_votes_on_proposal(...)`, voting power is spent according to the user parameters. While there are checks to ensure that the user has enough voting power to spend, the `spent_voting_power` storage variable is not updated after voting. The unchanged `spent_voting_power` is then written back to storage. This allows users to infinitely vote on a proposal in an infinite round without spending any votes. A snippet from the code is shown below:

```
fn _cast_votes_on_proposal(...) {
    // ...

    let mut spent_voting_power = _spent_voting_power::read((voter, proposal_id));

    // `spent_voting_power` is not updated or changed
    // ...

    ////////////////////////////////////////////////////////////////////////////////
    // @audit-issue The spent voting power is rewritten back to storage with no changes made
    //              Spent voting power never changes, users effectively have unlimited votes
    ////////////////////////////////////////////////////////////////////////////////
    _spent_voting_power::write((voter, proposal_id), spent_voting_power);

    // ...
}
```

Recommendation(s): Ensure that the `spent_voting_power` storage variable is updated to correctly track the amount of spent voting power per voter.

Status: Fixed

Update from the client: Fixed in d1c66f6. The spent_voting_power storage variable is now correctly updated.

6.3 [High] Timestamp to block number entries can be manipulated

File(s): ethereum_block.cairo

Description: The function `get_eth_block_number(...)` can retrieve an Ethereum block number for a given timestamp, using Herodotus to as the oracle for Ethereum block numbers. If this function is called with a timestamp that has already been stored, it returns that data. Otherwise, it will query Herodotus for the latest block number and associate that value with the provided timestamp. The function is shown below:

```
fn get_eth_block_number(timestamp: felt252) -> felt252 {
    let number = _timestamp_to_eth_block_number::read(timestamp);
    if number.is_non_zero() {
        number
    } else {
        ////////////////////////////////////////////////////////////////////////////////
        // @audit-issue If timestamp not associated with a block, current block is used
        //              A timestamp in the past or future can be linked to current block
        ////////////////////////////////////////////////////////////////////////////////
        let number = IL1HeadersStoreDispatcher {
            contract_address: _l1_headers_store::read()
        }.get_latest_l1_block();
        _timestamp_to_eth_block_number::write(timestamp, number);
        number
    }
}
```

Since on a call with a new `timestamp` argument the latest block number is used, if a `timestamp` argument from the past or future is used, it will still associate that timestamp with the current block number. This function is externally reachable through the function shown below, so any user can access this:

```
#[external]
fn get_eth_block_number(timestamp: felt252) -> felt252 {
    EthereumBlockRegistry::get_eth_block_number(timestamp)
}
```

This function is used to determine Ethereum ERC20 token balances for governance style proposal voting, which an attacker may take advantage of to affect voting and round outcomes. Since the voting strategies and snapshot timestamps are known when a round is deployed, an attacker could accumulate tokens to affect their voting power for a few blocks, and then call `get_eth_block_number(...)` with the timestamp matching the snapshot timestamp on L2. The attacker could then sell tokens to diminish their exposure, but maintain their high voting power since the future timestamp to be used will point to their current amount held. This could also be used to reduce the votes of other users, as any tokens that other users acquire to gain more voting power after the attacker has maliciously called `get_eth_block_number(...)` will not be used for vote weighting.

Recommendation(s): Consider preventing the function `get_eth_block_number(...)` from being called by any user, so the function can only be called internally with safe timestamp values retrieved from L1.

Status: Fixed

Update from the client: Addressed in bc63398. This commit removes the ability to populate the block for a future timestamp. An attacker can still populate a timestamp with a block in the registry between the timestamp in question and the first time `get_eth_block_number` is called by a round, but the risk is greatly reduced.

Nethermind comment: After further investigation there is no risk of the protocol reading a past timestamp that has not already been written to. Therefore reading past timestamps are not a concern and since future timestamps have been addressed, this issue can be considered fixed as all risks have been addressed.

6.4 [Medium] An infinite number of proposals can be created

File(s): timed/round.cairo

Description: In the `propose(...)` method on timed rounds, if the `config.proposal_threshold` is zero then any user can submit any number of proposals. The `propose` function is shown below:

```
fn propose(
    proposer: EthAddress,
    metadata_uri: Array<felt252>,
    used_proposing_strategies: Array<UserStrategy>,
) {
    // ...

    ////////////////////////////////////////////////////
    // @audit-issue No limit to the number of proposals
    ////////////////////////////////////////////////////
    assert(
        cumulative_proposition_power >= config.proposal_threshold.into(),
        'TR: Proposition power too low'
    );
    let proposal_id = _proposal_count::read() + 1;

    // ...

    _proposal_count::write(proposal_id);

    // ...
}
```

During round finalization, all proposals are passed through multiple loops. A malicious actor could propose a large amount of proposals to cause the Cairo step limit to be reached before execution can finish.

Recommendation(s): Consider limiting the number of proposals that can be created by each user.

Status: Fixed

Update from the client: Fixed in 14cdba6. Rather than retrieve and sort all active proposals during finalization, the top `winner_count` proposals are stored in a min heap, which is updated following each vote. This socializes the cost of maintaining an up-to-date list of the leading proposals. During finalization, we loop through the min heap, popping the root proposal in each iteration to efficiently determine winners. The min heap is packed into storage, ensuring that vote costs remain reasonable. Even with 25 winners, only four storage slots are used.

6.5 [Medium] Highly voted proposals can be cancelled to increase weighting of remaining unspent votes

File(s): `starknet/src/rounds/*`

Description: When a proposal is cancelled, its votes are not returned and they remain spent. It is possible for a malicious user who has a highly voted proposal and holds unspent voting power to cancel their proposal, removing all associated votes from the round winner calculations. This effectively increases the weighting of all unspent voting power, including the malicious user's.

This attack approach can be applied to a round with multiple winners. If enough votes can be removed upon cancellation of the proposal and the attacker's unspent voting power is high enough, it is possible for the attacker to create multiple new proposals and use the increased voting weight to ensure wins for these multiple proposals. When these conditions are met, the proposer can extract more value out of the round than the expected single award.

An example scenario is presented below:

A round exists where there is 1000 total voting power across all possible voters. Alice has a proposal which has been very well received with 500 votes. Of the remaining 500 voting power, Alice holds 250 unspent, and the other 250 votes are spent across 10 proposals at 25 votes each. There can be five award winners, winning 5 ETH each. Alice has the highest voted proposal and is guaranteed to win 5 ETH. However if she cancels her proposal the 500 votes on her proposal are gone, meaning that the winner calculations will only be based on the 500 remaining voting power, 250 of which still belongs to Alice as unspent, and the remaining spent. Alice creates five proposals and distributes 50 votes each, so now all of her proposals have a higher vote than the others at 25 each. When the round is finalized Alice will receive 25 ETH in awards, instead of the expected 5 ETH.

Recommendation(s): Consider preventing proposals from being cancelled during the voting period.

Status: Fixed

Update from the client: Fixed in 21cede3. Proposals can no longer be cancelled during the voting period.

6.6 [Medium] Incremental Merkle subtree is not updated upon proposal approval

File(s): infinite/round.cairo

Description: In the function `_approve_proposal(...)` when when adding a new proposal to the incremental Merkle tree, the current sub-tree is read from storage and a new leaf containing the approved proposal is appended to the tree. However, the resulting sub-tree with the appended leaf is not written back to storage. After inserting the first leaf, this causes all subsequent entries to produce incorrect Merkle roots. The function is shown below:

```
fn _approve_proposal(proposal_id: u32, ref proposal: Proposal) {
    // ...

    ////////////////////////////////////////////////////////////////////////
    // @audit-note: The incremental tree gets created here using the previous
    //              subtree information
    ////////////////////////////////////////////////////////////////////////
    let mut incremental_merkle_tree = IncrementalMerkleTreeTrait::<u256>::new(
        MAX_WINNER_TREE_DEPTH, winner_count, _read_sub_trees_from_storage(),
    );

    ////////////////////////////////////////////////////////////////////////
    // @audit-note: The hash of the leaf is calculated
    ////////////////////////////////////////////////////////////////////////
    let leaf = _compute_leaf(proposal_id, proposal);

    ////////////////////////////////////////////////////////////////////////
    // @audit-issue: The  new root is then calculated, but the new sub tree is
    //               not stored back in permanent storage.
    ////////////////////////////////////////////////////////////////////////
    _winner_merkle_root::write(incremental_merkle_tree.append_leaf(leaf));

    // ...
}
```

If there is more than one approved proposal in the tree, this will cause the incremental Merkle tree root to be corrupted, and no proposers will be able to claim awards.

Recommendation(s): Consider adding a call to `write_sub_trees_to_storage(...)` using `incremental_merkle_tree.sub_trees` as an argument after appending the new leaf.

Status: Fixed

Update from the client: Fixed in c604fc6. Sub-trees are now correctly written to storage and have been optimized. Now, only used merkle tree depths are written to storage.

6.7 [Medium] Spending voting power with multiple voting parameters are miscalculated

File(s): timed/round.cairo

Description: Voting on more than one proposal in the same transaction leads to users not being able to spent all their voting power. The calculations for `remaining_voting_power` are incorrect, as the cumulative amount spent is subtracted from the remaining voting power on each loop. The function is shown below:

```
fn _cast_votes_on_one_or_more_proposals(
    voter: EthAddress,
    cumulative_voting_power: u256,
    mut proposal_votes: Span<ProposalVote>
) {
    let mut spent_voting_power = _spent_voting_power::read(voter);
    let mut remaining_voting_power = cumulative_voting_power - spent_voting_power;
    loop {
        match proposal_votes.pop_front() {
            Option::Some(proposal_vote) => {
                ////////////////////////////////////////////////////////////////////////////
                // @audit-issue : Assuming voting to 4 proposals with power of (20,30,30,20).
                // total 100
                // spent = 0, remaining = 100 as initial values.
                // 1st iter spent_voting_power = 20, remaining_voting_power = 100 - 20 = 80
                // 2nd iter spent_voting_power = 50, remaining_voting_power 80 - 50 = 30 which is wrong !
                ////////////////////////////////////////////////////////////////////////////
                spent_voting_power += _cast_votes_on_proposal(
                    voter, *proposal_vote, remaining_voting_power,
                );
                remaining_voting_power -= spent_voting_power;
            },
            Option::None(_) => {
                // Update the spent voting power for the user
                _spent_voting_power::write(voter, spent_voting_power);
                break;
            },
        };
    };
}
```

Recommendation(s): Consider calculating the remaining voting power as follows:

```
remaining_voting_power = cumulative_voting_power - spent_voting_power;
```

Status: Fixed

Update from the client: Fixed in a41a825. The recommended remaining voting power calculation has been implemented.

6.8 [Medium] The `startTimestamp` in the L2 message payload is not modified when creating an infinite round

File(s): `InfiniteRound.sol`

Description: When creating an infinite round, the round creator passes a `RoundConfig` struct containing important parameters. The `startTimestamp` parameter is a part of this config data, and it is used to determine at what timestamp a voter's L1 ERC20 token balance was when a governance voting strategy is used.

The `InfiniteRound` contract has logic in the function `_register(...)` to ensure that the provided `startTimestamp` is not in the past. If the config's `startTimestamp` is in the past, then the current time should be used. This check is presented below:

```
function _register(RoundConfig memory config) internal {
    //...

    startTimestamp = _max(config.startTimestamp, uint40(block.timestamp));

    //...
}
```

This validation only changes the `startTimestamp` storage variable on the L1 round contract, and the `config.startTimestamp` value is left unchanged, then passed to L2 where the incorrect `startTimestamp` is used for voting power calculations. This could be exploited by a malicious user with permissions to create a round. They could use a timestamp from the past where they had more voting tokens than they currently have, or use a timestamp where other potential voters have less tokens to reduce their impact on the outcome of the round.

Recommendation(s): Ensure that the L1 and L2 track the same `startTimestamp` by updating `config.startTimestamp` to be the correct value before being passed to L2.

Status: Fixed

Update from the client: Fixed in 37a26ef. Both `config.startTimestamp` and `startTimestamp` are now set to the same value.

6.9 [Low] Assets cannot be reclaimed in infinite round while state is in `FinalizationPending`

File(s): InfiniteRound.sol

Description: For an infinite round, when the finalization process begins with `startFinalization(...)` the contract state is changed to `FinalizationPending`. During this time, while waiting for a finalize message from Starknet, it is not possible to cancel the round or recover funds. If the message cannot reach Starknet, either because the `msg.value` during message send was not high enough, or the Starknet sequencers are behaving incorrectly, it is not possible to cancel the round or reclaim funds.

To cancel a round or reclaim funds the contract state must be `Active`, `Finalized` or `Cancelled`. When the state is `FinalizationPending` these functions will revert. This issue only applies to infinite rounds, as timed rounds do not have a `FinazilationPending` state.

```
function _canReclaim() internal view returns (bool) {
    ////////////////////////////////////////////////////////////
    // @audit-issue Cannot claim if state is FinalizationPending
    ////////////////////////////////////////////////////////////
    if (state == RoundState.Cancelled) {
        return true;
    }
    if (state == RoundState.Finalized) {
        return block.timestamp - finalizedAt >= RECLAIM_UNCLAIMED_ASSETS_AFTER
          || claimedWinnerCount == currentWinnerCount;
    }
    return false;
}

function startFinalization() external payable onlyRoundManager {
    ////////////////////////////////////////////////////////////
    // @audit-issue Once finalization starts cannot recover any
    //              funds until finalize message from Starknet
    ////////////////////////////////////////////////////////////
    if (state != RoundState.Active || currentWinnerCount == 0) {
        revert FINALIZATION_NOT_AVAILABLE();
    }

    // ...
}
```

Recommendation(s): Consider adding a safety measure to allow funds to be retrieved from a `FinalizationPending` state if a reasonable amount of time has passed without the state changing to `Finalized`. Message cancellation for the `FINALIZE_ROUND` message from L1 -> L2 is an additional approach that can be considered.

Status: Fixed

Update from the client: Fixed in d285cb8. Funds can now be retrieved after one week has passed and the round is still in the FinalizationPending state.

6.10 [Low] Users can't re-vote on updated proposals

File(s): infinite/round.cairo

Description: When a proposal is updated, the `voting_power_for` storage variable is reset to zero, however the spent voting power for previous voters is not reset. This means that users who had previously voted on the proposal before the update will not be able to re-vote on the same or any other proposal, effectively disregarding their votes. Relevant code snippets are shown below:

```
fn edit_proposal(...) {
    // ...

    // @audit-issue: Proposal's "for" votes are cleared, but users spent voting power is not
    proposal.voting_power_for = 0;

    // ...
}
```

```
fn _cast_votes_on_proposal(...) {
    // ...

    // @audit-issue: The spent voting power doesn't depend on the version of the proposal
    //               "For" votes are reset on version updates, but the voting power is not
    let mut spent_voting_power = _spent_voting_power::read((voter, proposal_id));
    let mut remaining_voting_power = cumulative_voting_power - spent_voting_power;

    // ...
}
```

Recommendation(s): Consider returning the spent votes back to users who have voted for a proposal which has been updated.

Status: Fixed

Update from the client: Fixed in 364155a. Previously, to prevent infinite round proposers from bypassing proposal rejections, 'against' votes accumulated across proposal versions. Now, both 'for' and 'against' votes are reset when a proposal is edited. If proposers try to sidestep the 'against' vote threshold by modifying the proposal, voters can let the proposal go stale.

6.11 [Info] Array storage limit not checked in implementation

File(s): storage.cairo

Description: The write_span function used in StorageAccess::< Span<T> > doesn't check the size of the value Span. At most 256 field elements can be stored in a single storage variable. This function will fail if value.len() + offset > 256

```
fn write_span<T, impl TCopy: Copy<T>, impl TDrop: Drop<T>, impl TSA: StorageAccess<T>>(
    address_domain: u32, base: StorageBaseAddress, mut offset: u8, mut value: Span<T>
) -> SyscallResult<()> {
    ////////////////////////////////////////////////////////////////////////////////
    // @audit-issue: This call will fail if 'offset + value.len()' > 256
    ////////////////////////////////////////////////////////////////////////////////
    StorageAccess::<u32>::write_at_offset_internal(address_domain, base, offset, value.len())?;

    loop {
        match value.pop_front() {
            // ...
        };
    }
}
```

Recommendation(s): The size of the Span should be checked explicitly prior to writing it to storage in the StorageAccess impl. The result of value.len() + offset must be lower or equal than 255.

Status: Fixed

Update from the client: Added in d07108b.

6.12 [Info] Assets can be deposited to round during FINALIZED, FINALIZED_PENDING and CANCELLED state

File(s): PropHouse.sol

Description: It is possible to fund rounds through the function PropHouse.depositTo(...) after a round has been created. This allows for communities to collaboratively fund a round rather than needing a single address to provide all assets up-front during round creation. However, it is possible to use the depositTo function outside to deposit assets to a round outside of the ACTIVE state. If the round state is FINALIZED, FINALIZED_PENDING or CANCELLED the added funds will serve no purpose as all funds should already have been added to the round at that point.

It should be noted that this does not lead to lost funds, since the the depositor will be able to use the round functions reclaim(...) or reclaimTo(...) after enough time has passed.

Recommendation(s): Consider preventing asset deposits when the round state is not ACTIVE.

Status: Acknowledged

Update from the client: Acknowledged, but do not plan to fix as funds can always be reclaimed at a later time.

6.13 [Info] Incorrect comment related to function selector

File(s): Constants.sol

Description: In constants.sol the function selector for the Starknet l1_handler function route_call_to_round(...) is declared. The comment describing how selector was derived states "round_call_from_round" where it should actually be "route_call_from_round". The actual value being used is correct however, so L1 -> L2 messages will operate correctly. The relevant code is shown below:

```
// print(get_selector_from_name("round_call_to_round"))
uint256 constant ROUTE_CALL_TO_ROUND = 0x24931ca109ce0ffa87913d91f12d6ac327550c015a573c7b17a187c29ed8c1a;
```

Recommendation(s): Consider changing the comment to accurately reflect what the value in ROUTE_CALL_TO_ROUND represents.

Status: Fixed

Update from the client: Fixed in f726550.

6.14 [Info] Protocol does not support message cancellation

File(s): `ethereum/*`

Description: The Ethereum implementation of the protocol does not support message cancellation. When sending a message from L1 to L2 through `starknet.sendMessageToL2(...)` the caller must provide some `msg.value` which the StarkNet sequencers will accept as payment to execute the message on StarkNet. In the case that the `msg.value` is not enough for a sequencer to accept, these messages will remain unconsumed. Since it is not possible to cancel a message and receive the `msg.value` back, this could lead to permanently lost Ether. It should be noted that for a sequencer to ignore a message, the amount of Ether would have to be reasonably small.

Recommendation(s): Consider allowing message cancellation to ensure Ether cannot be lost if a sequencer does not accept a message.

Status: Acknowledged

Update from the client: Acknowledged, but do not plan to fix as the risk is low, the funds at risk are small, and the fix requires careful consideration as to not break other parts of the round.

6.15 [Info] `SpanStorageAccess::size_internal` can infinitely recurse

File(s): `storage.cairo`

Description: The implementation of the `StorageAccess<Span<T>>` trait named `SpanStorageAccess` has a `size_internal` function which calls itself infinitely. This method is not used in the protocol, but any dependency on it will result in failed transactions due to infinite loops.

```
    fn size_internal(value: Span<T>) -> u8 {
        ////////////////////////////////////////////////////////////////////////////////
        // @audit-issue: calling SpanStorageAccess::<T>::size_internal
        //               instead of StorageAccess::<T>::size_internal
        ////////////////////////////////////////////////////////////////////////////////
        1 + (SpanStorageAccess::<T>::size_internal(value) * downcast(value.len()).unwrap())
    }
```

Recommendation(s): If the Span is empty, return 1. If it's not, get the size of `T` by using `StorageAccess::<T>` instead of `SpanStorageAccess`.

Status: Fixed

Update from the client: Fixed in bf8a615.

6.16 [Best Practices] Avoid using `downcast` directly

File(s): `storage.cairo`

Description: The `downcast` extern function is used to convert `length` from `u32` to `u8`. A good practice is to use higher-level traits for type conversions, such as `TryInto` and `Into`, instead of using `upcast` and `downcast`.

```
fn read_span<T, impl TCopy: Copy<T>, impl TDrop: Drop<T>, impl TSA: StorageAccess<T>>(
    address_domain: u32, base: StorageBaseAddress, mut offset: u8
) -> SyscallResult<Span<T>> {
    let length = StorageAccess::<u32>::read_at_offset_internal(address_domain, base, offset)?;
    offset += 1; // Increment offset by 1 for the length.

    ////////////////////////////////////////////////////////////////////////////////
    // @audit-issue: Avoid using 'downcast' directly and prefer
    //               high-level 'TryInto' and 'Into' traits.
    ////////////////////////////////////////////////////////////////////////////////
    let exit_at = downcast(length).unwrap() + offset;
    let mut arr = Default::<Array<T>>::default();
}
```

Recommendation(s): Use the `try_into` method from the `TryInto` trait to perform downcasts. To avoid downcasting, `length` can also be directly stored into a `u8` using `StorageAccess::<u8>` instead, given that at most 255 elements can be stored in a storage variable.

Status: Fixed

Update from the client: Updated in 8f3806f.

6.17 [Best Practices] Ethereum addresses stored on Starknet should be type `EthAddress`

File(s): `ethereum_tx.cairo`, `ethereum_tx.cairo`, `commit_receiver.cario`

Description: Commits are received from Ethereum to Starknet through the `_commit_address` storage variable, which is declared in `commit_receiver.cairo`. The type for this storage variable is `ContractAddress`, which is used to store the address of a Starknet contract. The issue is shown in the code snippet below:

```
/// Initializes the contract by setting the commit address.
fn initializer(commit_address: ContractAddress) {
    _commit_address::write(commit_address);
}
```

Recommendation(s): Since `_commit_address` will be storing an Ethereum address, consider using the type `EthAddress` instead of `ContractAddress`.

Status: Fixed

Update from the client: Updated in 52ff5f0. While the `commit_address` parameter type has been changed to `EthAddress`, the `_commit_address` storage variable type has been changed to `felt252` to avoid the need to create a `StorageAccess` implementation for `EthAddress`.

6.18 [Best Practices] Function `get_power` can have a `view` visibility

File(s): `allowlist.cairo`

Description: The function `get_power(...)` is marked as `external`, but it doesn't mutate the state of any contract. It can be marked as `view` instead.

```
////////////////////////////////////////////////////////
// @audit-issue: This function can be marked as 'view'
////////////////////////////////////////////////////////
#[external]
fn get_power(
    timestamp: u64, user: felt252, params: Span<felt252>, user_params: Span<felt252>,
) -> u256 {
    AllowlistGovernancePowerStrategy::get_power(timestamp, user, params, user_params)
}
```

Recommendation(s): Consider marking the function as `#[view]`

Status: Fixed

Update from the client: Updated in 70fa82d.

6.19 [Best Practices] Function `get_slot_key(...)` cannot query a mapping with a key higher than `felt252` size

File(s): `single_slot_proof.cairo`

Description: The function `get_slot_key(...)` is used to calculate the storage slot for a mapping given a mapping slot and key in an Ethereum smart contract. However, this function is not able to access all potential storage slots for a mapping. The function takes two arguments `slot_index` and `mapping_key`, which represent the mapping slot and index respectively. The argument `mapping_key` is of type `felt252`, which means it is not possible to query for storage slots where the index would exceed what a `felt252` type can store. The function is shown below:

```
//////////////////////////////////////////////////////////////////////////////
// @audit-issue The key is felt252 but mapping key can be uint256 on Ethereum
//////////////////////////////////////////////////////////////////////////////
fn get_slot_key(slot_index: felt252, mapping_key: felt252) -> u256 {
    let mut encoded_array = Default::default();
    encoded_array.append(mapping_key.into());
    encoded_array.append(slot_index.into());

    keccak_u256s_be(encoded_array.span())
}
```

This means that it is not possible to query mapping entries where the key is greater that what a `felt252` type can represent. Some examples of mappings keys that may be affected are hashed data or signatures. Currently this does not cause a problem as the protocol only uses this function to query ERC20 balances, where the key is a 160 bit Ethereum address. However if this function is used elsewhere it may not behave correctly.

Recommendation(s): Consider changing `get_slot_key(...)` such that it can calculate the storage slot for all possible keys in a mapping.

Status: Fixed

Update from the client: Updated in 76aa0f7.

6.20 [Best Practices] Function `get_slot_value(...)` reverts if Ethereum storage slot has value `0`

File(s): `single_slot_proof.cairo`

Description: The function `get_slot_value(...)` is used to read the value in a given storage slot on Ethereum using Herodotus. Part of the logic in `get_slot_value` state that the transaction should revert if the loaded storage value is `0`, as shown in the code snippet below:

```
fn get_slot_value(
    timestamp: u64, user: felt252, params: Span<felt252>, user_params: Span<felt252>,
) -> u256 {

    // ...

    assert(slot_value.is_non_zero(), 'SSP: Slot value is zero');

    slot_value
}
```

Reverting when the storage value is zero should not be part of this function, as there may be cases when it is valid for storage to be zero. Instead, if a revert is necessary this should belong in the logic of the function caller, as there are valid use cases for `get_slot_value` returning zero. One such example could be querying a boolean mapping where different logic will execute depending on if the value is `true` or `false`..

Recommendation(s): Consider allowing `get_slot_value(...)` to return a value of zero, and if a revert is necessary then consider placing it in the calling logic instead.

Status: Fixed

Update from the client: Updated in f8d88fc.

6.21 [Best Practices] Public functions can be external

File(s): `contracts/*`

Description: It is considered a best practice to use the best-fitting visibility for each function. The following is a list of functions that have a visibility of `public` but can be changed to `external`:

```
CommunityHouse.isRound()
CreatorPassIssuer.uri()
AssetRound.uri()
ERC1155.uri()
```

Recommendation(s): Consider changing the visibility of the functions listed below to `external`.

Status: Fixed

Update from the client: Updated in c41e7b4.

6.22 [Best Practices] Unnecessary mutable variables/arguments

File(s): `contracts/starknet/*`

Description: The following variables and arguments have the `mut` keyword but are not modified.

```
_computer_root(...)
    sub_tree

parse_strategies(...)
    starting_index

_mergesort_proposals_by_voting_power_desc_and_slice(...)
    left_arr
    right_arr
    sorted_left
    sorted_right

_cast_votes_on_proposal(...)
    spent_voting_power
    remaining_voting_power
```

Note that for the function `_cast_votes_on_proposal(...)` some of the listed variables should keep the `mut` keyword after other issues in the report have been addressed, where they should be updated but currently are not.

Recommendation(s): Consider removing the `mut` keyword for the variables and arguments listed above.

Status: Fixed

Update from the client: Removed in cf72793.

6.23 [Best Practices] Unnecessary setting of `state` in round contracts

File(s): `ethereum/rounds/*`

Description: The `TimedRound` and `InfiniteRound` contracts have a storage variable `state` which tracks what the current state of the round is. When initializing these rounds, the function `_register(...)` sets `state` to `RoundState.Active` as shown below:

```
function _register(RoundConfig memory config) internal {
    //...

    state = RoundState.Active;

    // ...
}
```

The definition of the `RoundState` enum in both `ITimedRound` and `IInfiniteRound` set `Active` as the first item. The first item in an enum has a value of zero, and since the contract has just been deployed `state` would be zero as it hasn't been set yet. This means that setting `state` to `RoundState.Active` again is an unnecessary SSTORE operation.

Recommendation(s): Removing the unnecessary setting of `state` in `_register` for round contracts. Alternatively this can be kept for code readability purposes, at the cost of a slight gas increase.

Status: Fixed

Update from the client: Removed in ff91a94 and added a comment noting that the round state defaults to active.

6.24 [Best Practices] Unnecessary snap and desnap operations

File(s): `strategy.cairo`

Description: The `_compute_strategy_id` function takes a snapshot of a `Strategy` as a parameter and desnaps it right away. The `strategy` can simply be passed by value instead, as `Strategy` derives the `Copy` trait.

```
fn register_strategy_if_not_exists(mut strategy: Strategy) -> felt252 {
    // ...

    let strategy_id = _compute_strategy_id(@strategy);

    // ...

}

fn _compute_strategy_id(strategy: @Strategy) -> felt252 {
        ////////////////////////////////////////////////////////////////////////////////
        // @audit-issue: 'strategy' is passed by snapshot and directly de-snapped.
        // It can be passed by value instead.
        ////////////////////////////////////////////////////////////////////////////////
        let mut strategy = *strategy;

        // ...
}
```

Recommendation(s): Consider passing the `strategy` parameter as `mut strategy: Strategy` instead.

Status: Fixed

Update from the client: Removed in 8db19c8.

6.25 [Best Practices] Unnecessary variable `i` in `get_cumulative_governance_power(...)`

File(s): `libraries/round.cairo`

Description: The function `get_cumulative_governance_power(...)` has a variable `i` which is set and incremented, but is never used. A snippet from the function is shown below:

```
fn get_cumulative_governance_power(...) -> u256 {
    let mut i = 0;

    // ...
    loop {
        match used_strategies.pop_front() {
            Option::Some(s) => {
                // ...
                i += 1;
                // ...
            },
            Option::None(_) => {
                break;
            },
        };
    };
    // ...
}
```

Recommendation(s): Consider removing the unused variable.

Status: Fixed

Update from the client: Removed in 27ca7ae.

7 Documentation Evaluation

Software documentation refers to the written or visual information that describes the functionality, architecture, design, and implementation of software. It provides a comprehensive overview of the software system and helps users, developers, and stakeholders understand how the software works, how to use it, and how to maintain it. Software documentation can take different forms, such as user manuals, system manuals, technical specifications, requirements documents, design documents, and code comments. Software documentation plays a critical role in software development, as it enables effective communication between developers, testers, users, and other stakeholders involved in the software development process. It helps to ensure that everyone involved in the development process has a shared understanding of the software system and its functionality. Moreover, software documentation can improve software maintenance by providing a clear and complete understanding of the software system, making it easier for developers to maintain, modify, and update the software over time. Smart contracts can use various types of software documentation. Some of the most common types include:

- Technical whitepaper: A technical whitepaper is a comprehensive document that describes the design and technical details of the smart contract. It includes information about the purpose of the contract, its architecture, its components, and how they interact with each other;
- User manual: A user manual is a document that provides information about how to use the smart contract. It includes step-by-step instructions on how to perform various tasks and explains the different features and functionalities of the contract;
- Code documentation: Code documentation is a document that provides details about the code of the smart contract. It includes information about the functions, variables, and classes used in the code, as well as explanations of how they work;
- API documentation: API documentation is a document that provides information about the API (Application Programming Interface) of the smart contract. It includes details about the methods, parameters, and responses that can be used to interact with the contract;
- Testing documentation: Testing documentation is a document that provides information about how the smart contract was tested. It includes details about the test cases that were used, the results of the tests, and any issues that were identified during testing;
- Audit documentation: Audit documentation includes reports, notes, and other materials related to the security audit of the smart contract. This type of documentation is critical in ensuring that the smart contract is secure and free from vulnerabilities.

These types of documentation are essential for smart contract development and maintenance. They help ensure that the contract is properly designed, implemented, and tested, and they provide a reference for developers who need to modify or maintain the contract in the future.

Nouns DAO has provided the documentation for Prop-House in the form of a README.md. The quality of this documentation is very high, and provides detailed information on the following:

- A general overview of the concepts of the protocol;
- Descriptions for each major contract;
- Necessary environment setup and dependencies;
- Instructions on how to build contracts and run test suite.

The quality of the inline comments for both Solidity and Cairo contracts are of very high quality. In Solidity files, each function has an associated NatSpec comment explaining the function's purpose, inputs and outputs. For Cairo files, an equivalent explanation of purpose, inputs and outputs has been provided as well. Aside from general function comments, specific lines of code that may be hard for a reader to understand are accompanied with an explanation.

8 Test Suite Evaluation

8.1 Contracts Compilation

8.1.1 Solidity

```
> yarn build:l1

yarn run v1.22.19
$ FOUNDRY_PROFILE=ignore_test forge build
[] Compiling...
[] Compiling 85 files with 0.8.20
[] Solc 0.8.20 finished in 6.35s
Compiler run successful!
Done in 7.55s.
```

8.1.2 Cairo

```
> yarn build:l2

yarn run v1.22.19
$ cd contracts/starknet && scarb build
   Compiling prop_house v0.1.0
    → (/home/NM-0096/prop-house-monorepo/packages/prop-house-protocol/contracts/starknet/Scarb.toml)
    Finished release target(s) in 10 seconds
Done in 10.24s.
```

8.2 Tests Output

8.2.1 Solidity

```
> yarn test:l1

Running 12 tests for test/ethereum/InfiniteRound.t.sol:InfiniteRoundTest
[PASS] test_claim() (gas: 420345)
[PASS] test_claimWithoutRootReverts() (gas: 353767)
[PASS] test_duplicateClaimReverts() (gas: 414646)
[PASS] test_noAgainstQuorumReverts() (gas: 162539)
[PASS] test_noForQuorumReverts() (gas: 162479)
[PASS] test_noProposingStrategiesWhenProposalThresholdSetReverts() (gas: 162542)
[PASS] test_noVotingStrategiesReverts() (gas: 162331)
[PASS] test_reclaimDuringActiveRoundReverts() (gas: 345273)
[PASS] test_reclaimSucceedsOnCancel() (gas: 366960)
[PASS] test_reclaimToDuringActiveRoundReverts() (gas: 345450)
[PASS] test_reclaimWithoutDepositCreditsReverts() (gas: 386739)
[PASS] test_votePeriodDurationTooShortReverts() (gas: 162420)
Test result: ok. 12 passed; 0 failed; finished in 12.55ms

Running 8 tests for test/ethereum/Manager.t.sol:ManagerTest
[PASS] test_registerHouse() (gas: 49867)
[PASS] test_registerHouseNonOwnerReverts() (gas: 20730)
[PASS] test_registerRound() (gas: 52947)
[PASS] test_registerRoundNonOwnerReverts() (gas: 23100)
[PASS] test_unregisterHouse() (gas: 38939)
[PASS] test_unregisterHouseNonOwnerReverts() (gas: 20743)
[PASS] test_unregisterRound() (gas: 41876)
[PASS] test_unregisterRoundNonOwnerReverts() (gas: 23031)
Test result: ok. 8 passed; 0 failed; finished in 17.86ms
```

```
Running 19 tests for test/ethereum/TimedRound.t.sol:TimedRoundTest
[PASS] test_awardWinnerCountMismatchReverts() (gas: 176706)
[PASS] test_claim() (gas: 408305)
[PASS] test_claimFromWrongCallerReverts() (gas: 387445)
[PASS] test_claimWithWrongAmountReverts() (gas: 390024)
[PASS] test_claimWithWrongTokenReverts() (gas: 390032)
[PASS] test_claimWithoutRootReverts() (gas: 353733)
[PASS] test_duplicateClaimReverts() (gas: 397349)
[PASS] test_finalize() (gas: 383430)
[PASS] test_noProposingStrategiesWhenProposalThresholdSetReverts() (gas: 163869)
[PASS] test_noVotingStrategiesReverts() (gas: 163581)
[PASS] test_noWinnerCountReverts() (gas: 163684)
[PASS] test_reclaimDuringActiveRoundReverts() (gas: 347647)
[PASS] test_reclaimSucceedsOnCancel() (gas: 369626)
[PASS] test_reclaimToDuringActiveRoundReverts() (gas: 347845)
[PASS] test_reclaimWithoutDepositCreditsReverts() (gas: 389318)
[PASS] test_remainingProposalPeriodDurationTooShortReverts() (gas: 163609)
[PASS] test_splitAwardAmountNotMultipleOfWinnerCountReverts() (gas: 165777)
[PASS] test_votePeriodDurationTooShortReverts() (gas: 163674)
[PASS] test_winnerCountTooHighReverts() (gas: 163730)
Test result: ok. 19 passed; 0 failed; finished in 12.57ms


Running 36 tests for test/ethereum/PropHouse.t.sol:PropHouseTest
[PASS] test_batchDepositToInvalidRoundReverts() (gas: 40068)
[PASS] test_batchDepositToWithERC1155() (gas: 125767)
[PASS] test_batchDepositToWithERC20s() (gas: 127415)
[PASS] test_batchDepositToWithERC721() (gas: 123365)
[PASS] test_batchDepositToWithETHNoValue() (gas: 38982)
[PASS] test_batchDepositToWithExactETHAmount() (gas: 94196)
[PASS] test_batchDepositToWithExtraETH() (gas: 106293)
[PASS] test_batchDepositToWithManyAssets() (gas: 334498)
[PASS] test_createAndFundRoundOnExistingHouseWithERC1155() (gas: 358051)
[PASS] test_createAndFundRoundOnExistingHouseWithERC20s() (gas: 359731)
[PASS] test_createAndFundRoundOnExistingHouseWithERC721() (gas: 355629)
[PASS] test_createAndFundRoundOnExistingHouseWithETHNoValue() (gas: 185244)
[PASS] test_createAndFundRoundOnExistingHouseWithExactETHAmount() (gas: 326466)
[PASS] test_createAndFundRoundOnExistingHouseWithExtraETH() (gas: 348878)
[PASS] test_createAndFundRoundOnExistingHouseWithManyAssets() (gas: 566413)
[PASS] test_createAndFundRoundOnNewHouseWithERC1155() (gas: 484042)
[PASS] test_createAndFundRoundOnNewHouseWithERC20s() (gas: 485678)
[PASS] test_createAndFundRoundOnNewHouseWithERC721() (gas: 481594)
[PASS] test_createAndFundRoundOnNewHouseWithETHNoValue() (gas: 311109)
[PASS] test_createAndFundRoundOnNewHouseWithExactETHAmount() (gas: 452456)
[PASS] test_createAndFundRoundOnNewHouseWithExtraETH() (gas: 474880)
[PASS] test_createAndFundRoundOnNewHouseWithManyAssets() (gas: 692386)
[PASS] test_createRoundOnExistingHouse() (gas: 267310)
[PASS] test_createRoundOnExistingHouseNonExistentHouseReverts() (gas: 69329)
[PASS] test_createRoundOnExistingHouseUnregisteredRoundReverts() (gas: 37897)
[PASS] test_createRoundOnNewHouse() (gas: 393431)
[PASS] test_createRoundOnNewHouseUnregisteredHouseReverts() (gas: 74710)
[PASS] test_createRoundOnNewHouseUnregisteredRoundReverts() (gas: 46679)
[PASS] test_createRoundOnNewHouseWrongHouseImplReverts() (gas: 1016175)
[PASS] test_depositToInvalidRoundReverts() (gas: 39099)
[PASS] test_depositToWithERC1155() (gas: 120543)
[PASS] test_depositToWithERC20s() (gas: 122257)
[PASS] test_depositToWithERC721() (gas: 118183)
[PASS] test_depositToWithETHNoValue() (gas: 37020)
[PASS] test_depositToWithExactETHAmount() (gas: 88890)
[PASS] test_depositToWithExtraETH() (gas: 100967)
Test result: ok. 36 passed; 0 failed; finished in 17.91ms
Done in 16.23s.
```

8.2.2 Cairo

```
> yarn test:l2

yarn run v1.22.19
$ cd contracts/starknet && scarb run test
running 7 tests
test prop_house::test::infinite_round_test::test_infinite_round_decode_params ... ok
test prop_house::test::merkle_test::test_pedersen_merkle_tree ... ok
test prop_house::test::timed_round_test::test_timed_round_decode_params ... ok
test prop_house::test::merkle_test::test_keccak_incremental_merkle_tree_full_failure ... ok
test prop_house::test::merkle_test::test_keccak_merkle_tree ... ok
test prop_house::test::merkle_test::test_keccak_incremental_merkle_tree ... ok
test prop_house::test::timed_round_test::test_timed_round_get_n_proposals_by_voting_power_desc ... ok
test result: ok. 7 passed; 0 failed; 0 ignored; 0 filtered out;
Done in 11.33s.
```

8.2.3 Crosschain

```
> yarn test:crosschain

yarn run v1.22.17
$ hardhat test test/crosschain/**/*.test.ts --network 'ethereumLocal' --starknet-network 'starknetLocal'
Starknet plugin using the active environment.
Using network starknetLocal at http://127.0.0.1:5050/
Compiled 82 Solidity files successfully


  InfiniteRoundStrategy - ETH Signature Auth Strategy
     should create a proposal using an Ethereum signature (2607ms)
     should create a vote using an Ethereum signature (3885ms)
     should allow a winner to claim their award (1555ms)
     should finalize a round (708ms)

  InfiniteRoundStrategy - ETH Transaction Auth Strategy
     should create a proposal using an Ethereum transaction (2014ms)
     should not allow the same commit to be executed multiple times (2432ms)
     should fail if the correct hash of the payload is not committed on L1 before execution is called (570ms)
     should fail if the commit sender address is not equal to the address in the payload (655ms)
     should cancel a proposal using an Ethereum transaction (4109ms)
     should create votes using an Ethereum transaction (4531ms)
     should allow winners to claim their awards (1711ms)
     should finalize a round (735ms)

  TimedRoundStrategy - ETH Signature Auth Strategy
     should create a proposal using an Ethereum signature (2060ms)
     should create a vote using an Ethereum signature (2560ms)
     should finalize a round and allow a winner to claim their award (2230ms)

  TimedRoundStrategy - ETH Transaction Auth Strategy
     should create a proposal using an Ethereum transaction (1957ms)
     should not allow the same commit to be executed multiple times (2214ms)
     should fail if the correct hash of the payload is not committed on L1 before execution is called (585ms)
     should fail if the commit sender address is not equal to the address in the payload (538ms)
     should cancel a proposal using an Ethereum transaction (3772ms)
     should create a vote using an Ethereum transaction (2140ms)
     should finalize a round and allow the winner to claim their award (2206ms)


  22 passing (7m)
```

8.3 Tests Summary

The test suite for the Solidity contracts are well implemented and covers edge cases, exploring function calls with different inputs, both valid and invalid. The test suite for the Cairo contracts at the time of this audit is incomplete, only parameter decoding for rounds and the incremental Merkle tree implementation are tested. It is recommended to add more tests to cover the Cairo contracts to ensure that all code behaves as expected not only for the code in its current state, but in the future when changes are made. Integration tests are also implemented to ensure that communication between each chain behaves correctly. Functionality surrounding creating proposals, casting votes, signature validation, determining round winners, and claiming awards on the Ethereum layer

9 About Nethermind

Nethermind is a Blockchain Research and Software Engineering company. Our work touches every part of the web3 ecosystem - from layer 1 and layer 2 engineering, cryptography research, and security to application-layer protocol development. We offer strategic support to our institutional and enterprise partners across the blockchain, digital assets, and DeFi sectors, guiding them through all stages of the research and development process, from initial concepts to successful implementation.

We offer security audits of projects built on EVM-compatible chains and Starknet. We are active builders of the Starknet ecosystem, delivering a node implementation, a block explorer, a Solidity-to-Cairo transpiler, and formal verification tooling. Nethermind also provides strategic support to our institutional and enterprise partners in blockchain, digital assets, and decentralized finance (DeFi). In the next paragraphs, we introduce the company in more detail.

Blockchain Security: At Nethermind, we believe security is vital to the health and longevity of the entire Web3 ecosystem. We provide security services related to Smart Contract Audits, Formal Verification, and Real-Time Monitoring. Our Security Team comprises blockchain security experts in each field, often collaborating to produce comprehensive and robust security solutions. The team has a strong academic background, can apply state-of-the-art techniques, and is experienced in analyzing cutting-edge Solidity and Cairo smart contracts, such as ArgentX and StarkGate (the bridge connecting Ethereum and StarkNet). Most team members hold a Ph.D. degree and actively participate in the research community, accounting for 240+ articles published and 1,450+ citations in Google Scholar. The security team adopts customer-oriented and interactive processes where clients are involved in all stages of the work.

Blockchain Core Development: Our core engineering team, consisting of over 20 developers, maintains, improves, and upgrades our flagship product - the Nethermind Ethereum Execution Client. The client has been successfully operating for several years, supporting both the Ethereum Mainnet and its testnets, and now accounts for nearly a quarter of all synced Mainnet nodes. Our unwavering commitment to Ethereum's growth and stability extends to sidechains and layer 2 solutions. Notably, we were the sole execution layer client to facilitate Gnosis Chain's Merge, transitioning from Aura to Proof of Stake (PoS), and we are actively developing a full-node client to bolster Starknet's decentralization efforts. Our core team equips partners with tools for seamless node set-up, using generated docker-compose scripts tailored to their chosen execution client and preferred configurations for various network types.

DevOps and Infrastructure Management: Our infrastructure team ensures our partners' systems operate securely, reliably, and efficiently. We provide infrastructure design, deployment, monitoring, maintenance, and troubleshooting support, allowing you to focus on your core business operations. Boasting extensive expertise in Blockchain as a Service, private blockchain implementations, and node management, our infrastructure and DevOps engineers are proficient with major cloud solution providers and can host applications in-house or on clients' premises. Our global in-house SRE teams offer 24/7 monitoring and alerts for both infrastructure and application levels. We manage over 5,000 public and private validators and maintain nodes on major public blockchains such as Polygon, Gnosis, Solana, Cosmos, Near, Avalanche, Polkadot, Aptos, and StarkWare L2. Sedge is an open-source tool developed by our infrastructure experts, designed to simplify the complex process of setting up a proof-of-stake (PoS) network or chain validator. Sedge generates docker-compose scripts for the entire validator set-up based on the chosen client, making the process easier and quicker while following best practices to avoid downtime and being slashed.

Cryptography Research: At Nethermind, our Cryptography Research team is dedicated to continuous internal research while fostering close collaboration with external partners. The team has expertise across a wide range of domains, including cryptography protocols, consensus design, decentralized identity, verifiable credentials, Sybil resistance, oracles, and credentials, distributed validator technology (DVT), and Zero-knowledge proofs. This diverse skill set, combined with strong collaboration between our engineering teams, enables us to deliver cutting-edge solutions to our partners and clients.

Smart Contract Development & DeFi Research: Our smart contract development and DeFi research team comprises 40+ world-class engineers who collaborate closely with partners to identify needs and work on value-adding projects. The team specializes in Solidity and Cairo development, architecture design, and DeFi solutions, including DEXs, AMMs, structured products, derivatives, and money market protocols, as well as ERC20, 721, and 1155 token design. Our research and data analytics focuses on three key areas: technical due diligence, market research, and DeFi research. Utilizing a data-driven approach, we offer in-depth insights and outlooks on various industry themes.

Our suite of L2 tooling: Warp is Starknet's approach to EVM compatibility. It allows developers to take their Solidity smart contracts and transpile them to Cairo, Starknet's smart contract language. In the short time since its inception, the project has accomplished many achievements, including successfully transpiling Uniswap v3 onto Starknet using Warp.

- **Voyager** is a user-friendly Starknet block explorer that offers comprehensive insights into the Starknet network. With its intuitive interface and powerful features, Voyager allows users to easily search for and examine transactions, addresses, and contract details. As an essential tool for navigating the Starknet ecosystem, Voyager is the go-to solution for users seeking in-depth information and analysis;
- **Horus** is an open-source formal verification tool for StarkNet smart contracts. It simplifies the process of formally verifying Starknet smart contracts, allowing developers to express various assertions about the behavior of their code using a simple assertion language;
- **Juno** is a full-node client implementation for Starknet, drawing on the expertise gained from developing the Nethermind Client. Written in Golang and open-sourced from the outset, Juno verifies the validity of the data received from Starknet by comparing it to proofs retrieved from Ethereum, thus maintaining the integrity and security of the entire ecosystem.

Learn more about us at nethermind.io.

General Advisory to Clients

As auditors, we recommend that any changes or updates made to the audited codebase undergo a re-audit or security review to address potential vulnerabilities or risks introduced by the modifications. By conducting a re-audit or security review of the modified codebase, you can significantly enhance the overall security of your system and reduce the likelihood of exploitation. However, we do not possess the authority or right to impose obligations or restrictions on our clients regarding codebase updates, modifications, or subsequent audits. Accordingly, the decision to seek a re-audit or security review lies solely with you.

Disclaimer

This report is based on the scope of materials and documentation provided by you to Nethermind in order that Nethermind could conduct the security review outlined in **1. Executive Summary** and **2. Audited Files**. The results set out in this report may not be complete nor inclusive of all vulnerabilities. Nethermind has provided the review and this report on an as-is, where-is, and as-available basis. You agree that your access and/or use, including but not limited to any associated services, products, protocols, platforms, content, and materials, will be at your sole risk. Blockchain technology remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer, or any other areas beyond the programming language, or other programming aspects that could present security risks. This report does not indicate the endorsement of any particular project or team, nor guarantee its security. No third party should rely on this report in any way, including for the purpose of making any decisions to buy or sell a product, service or any other asset. To the fullest extent permitted by law, Nethermind disclaims any liability in connection with this report, its content, and any related services and products and your use thereof, including, without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement. Nethermind does not warrant, endorse, guarantee, or assume responsibility for any product or service advertised or offered by a third party through the product, any open source or third-party software, code, libraries, materials, or information linked to, called by, referenced by or accessible through the report, its content, and the related services and products, any hyperlinked websites, any websites or mobile applications appearing on any advertising, and Nethermind will not be a party to or in any way be responsible for monitoring any transaction between you and any third-party providers of products or services. As with the purchase or use of a product or service through any medium or in any environment, you should use your best judgment and exercise caution where appropriate. FOR AVOIDANCE OF DOUBT, THE REPORT, ITS CONTENT, ACCESS, AND/OR USAGE THEREOF, INCLUDING ANY ASSOCIATED SERVICES OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, INVESTMENT, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.