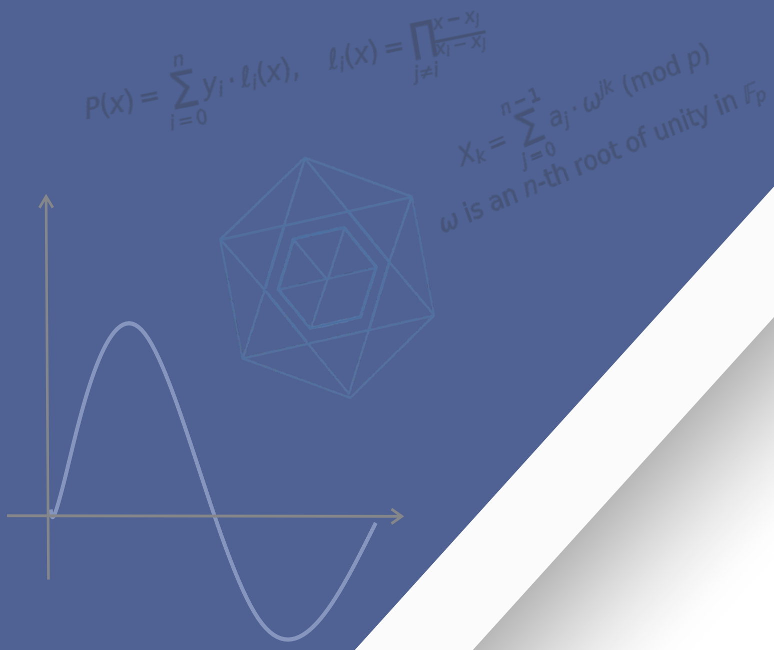

$P(x) = \sum_{i=0}^{n} y_i \cdot \ell_i(x), \quad \ell_i(x) = \prod_{j \neq i} \frac{x - x_j}{x_i - x_j}$
$X_k = \sum_{j=0}^{n-1} a_j \cdot \omega^{jk} \pmod p$
ω is an n-th root of unity in $\mathbb{F}_p$

Security Review Report

NM-0530 - Lido Accounting Zk Oracle built on Succint SP1

July 17, 2025

Contents

1 Executive Summary

This document presents the security review performed by Nethermind Security for the Lido Accounting SP1 Oracle smart contracts and zkVM circuit from Jun. 16, 2025 to Jul. 4, 2025. The Lido Accounting Oracle SP1 implementation is a circuit that verifies the correct reporting by the existing Lido oracles. This system aims to prove the change between two lido states represented in a report. The proof for the circuit is submitted on-chain and stored for the given reference slot. These proofs are then stored historically for each reference slot to be called when a negative rebase, according to LIP-23, has been detected.

The Lido Protocol is a staking platform that enables users to stake their assets on Ethereum, receiving liquid staking tokens in return, which can then be used across DeFi applications for additional yield and utility.

The audited code comprises 341 lines of code written in the Solidity language and 3,245 lines of code written in the Rust language. The audit included the SP1 Acccounting Report contract together with the zkVM circuit itself and deployment scripts.

The audit was performed using (a) manual analysis of the codebase and (b) automated analysis tools.

Along this document, we report nine points of attention, where one is classified as `High`, one is classified as `Medium`, one is classified as `Low`, and six are classified as `Informational` or `Best Practice`. **Fig. 1** summarizes the issues identified in *smart contracts and the circuit*. We detected *four issues in the deployment scripts*, which are summarized in Section 3 and described in detail in the Appendix A.

This document is organized as follows. Section 2 presents the files in the scope. Section 3 summarizes the issues. Section 4 presents the system overview. Section 5 discusses the risk rating methodology. Section 6 details the issues. Section 7 discusses the documentation provided by the client for this audit. Section 8 concludes the document. Appendix describes the review of the deployment scripts.

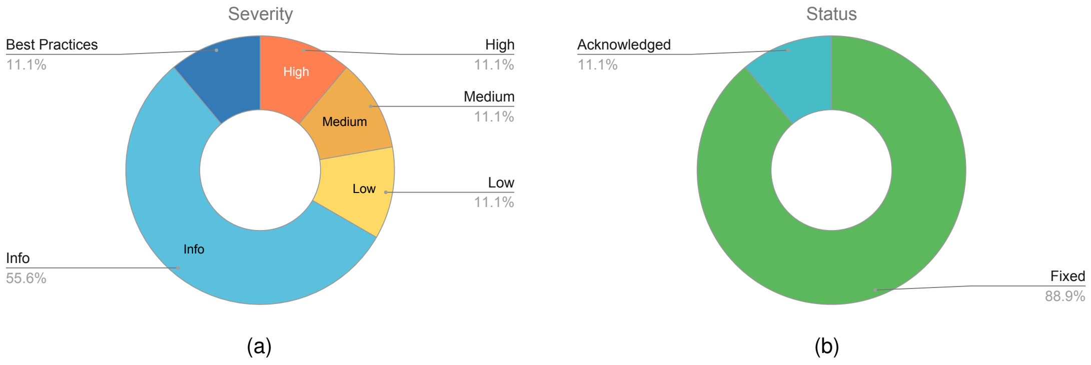


Fig. 1: Distribution of issues: Critical (0), **High** (1), **Medium** (1), **Low** (1), **Undetermined** (0), **Informational** (5), **Best Practices** (1). **Distribution of status: Fixed** (8), **Acknowledged** (1), **Mitigated** (0), **Unresolved** (0)

Summary of the Audit

Audit Type	Security Review
Initial Report	July 4, 2025
Final Report	July 17, 2025
Response from Client	Regular responses during audit engagement
Repository	sp1-lido-accounting-zk
Audit Commit	`303a0ef0484def3ab7c134f9a47e1ea353307008`
Final Commit	`06bb68acdc86b3722ccb4be726f271a7f26a3b59`
Documentation	lip-23, oracle-report-sanity-checker
Documentation Assessment	High

Summary of Engagement

Dates	Methods	Consultants Engaged	Level of Effort
Jun. 16-Jul. 4, 2025	Manual	5	15-person weeks

2 Audited Files

	Contract	LoC	Comments	Ratio	Blank	Total
1	contracts/src/ISecondOpinionOracle.sol	13	2	15.4%	1	16
2	contracts/src/PausableUntil.sol	72	14	19.4%	16	102
3	contracts/src/Sp1LidoAccountingReportContract.sol	256	67	26.2%	57	380
	Total	**341**	**83**	**24.3%**	**74**	**498**

	Circuit	LoC	Comments	Ratio	Blank	Total
1	crates/program/src/bin/main.rs	65	0	0%	12	77
2	crates/shared/src/lib/circuit_logic/input_verification.rs	545	26	4.77%	71	642
3	crates/shared/src/lib/circuit_logic/io.rs	142	1	0.7%	13	156
4	crates/shared/src/lib/circuit_logic/report.rs	70	2	3.33%	9	81
5	crates/shared/src/lib/circuit_logic.rs	3	0	0%	0	3
6	crates/shared/src/lib/io/eth_io.rs	388	0	0%	74	462
7	crates/shared/src/lib/io/program_io.rs	64	8	12.5%	10	82
8	crates/shared/src/lib/io/serde_utils.rs	71	0	0%	8	79
9	crates/shared/src/lib/eth_consensus_layer.rs	410	58	14.1%	67	535
10	crates/shared/src/lib/eth_execution_layer.rs	10	0	0%	2	12
11	crates/shared/src/lib/eth_spec.rs	17	0	0%	5	22
12	crates/shared/src/lib/io.rs	3	0	0%	0	3
13	crates/shared/src/lib/lib.rs	8	0	0%	0	8
14	crates/shared/src/lib/lido.rs	277	19	6.8%	39	335
15	crates/shared/src/lib/merkle_proof.rs	757	29	3.8%	127	913
16	crates/shared/src/lib/util.rs	14	4	28.6%	1	19
	Total	**2844**	**147**	**5.2%**	**438**	**3449**

	Scripts	LoC	Comments	Ratio	Blank	Total
1	justfile	91	17	18.7%	41	149
2	contracts/script/Deploy.s.sol	59	7	11.9%	11	77
3	crates/dev_script/src/bin/deploy.rs	74	14	18.9%	11	99
4	crates/dev_script/src/lib/scripts/deploy.rs	146	1	0.6%	18	165
5	crates/scripts/src/lib/deploy.rs	31	0	0%	4	35
	Total	**401**	**39**	**9.7%**	**85**	**536**

3 Summary of Issues

The table below presents a summary of issues detected in the contracts and circuits.

	Finding	Severity	Update
1	The `cl_balance` manipulation and oracle DOS through incorrect validator indices in `lido_changed`	High	Fixed
2	Arbitrary old state can be used	Medium	Fixed
3	No checks on new state	Low	Fixed
4	Inefficient checks for old lido state	Info	Acknowledged
5	`TimestampOutOfRange` error reports parameters in the wrong order	Info	Fixed
6	Redundant merkle proof check	Info	Fixed
7	Unnecessary data in `lido_changed`	Info	Fixed
8	Update crates/program rust-toolchain	Info	Fixed
9	Unchecked integer overflows	Best Practices	Fixed

The table below summarizes the issues identified in the script files:

	Finding	Update
10	Ambiguous Placeholder Usage for API Keys in Environment Configuration	Fixed
10	Inconsistency: Sensitivity Flag for Execution vs. Consensus Layer RPC	Fixed
11	Sensitive values are manually logged, ignoring the ‘sensitive‘ flag	Fixed
12	Current deploy command does not verify the deployed contract	Acknowledged

4 System Overview

In this section we present the Lido Accounting zk-Oracle and the relevant parts to the audit. For more in-depth discussion of the oracle please refer to LIP-23.

The zk-Oracle is a circuit written in SP1 that aims to prove and validate the reports submitted on-chain by Lido's oracles. The zk-Oracle is to be used as a second-opinion when the Lido state requires verification when there has been a negative rebase as defined in LIP-23.

Traditionally, Lido has an approved set of 9 oracles that agree on the state of the Lido validators by performing a hash consensus. This hash consensus is a multi-sig with 5-out-of-9 needed to agree on the status of the beacon chain. Specifically, this hash consensus contains all the relevant beacon chain information that is used to define the status of Lido validators. To verify this consensus, Lido is adding a zk-Oracle. The overall flow is shown in Figure 2.

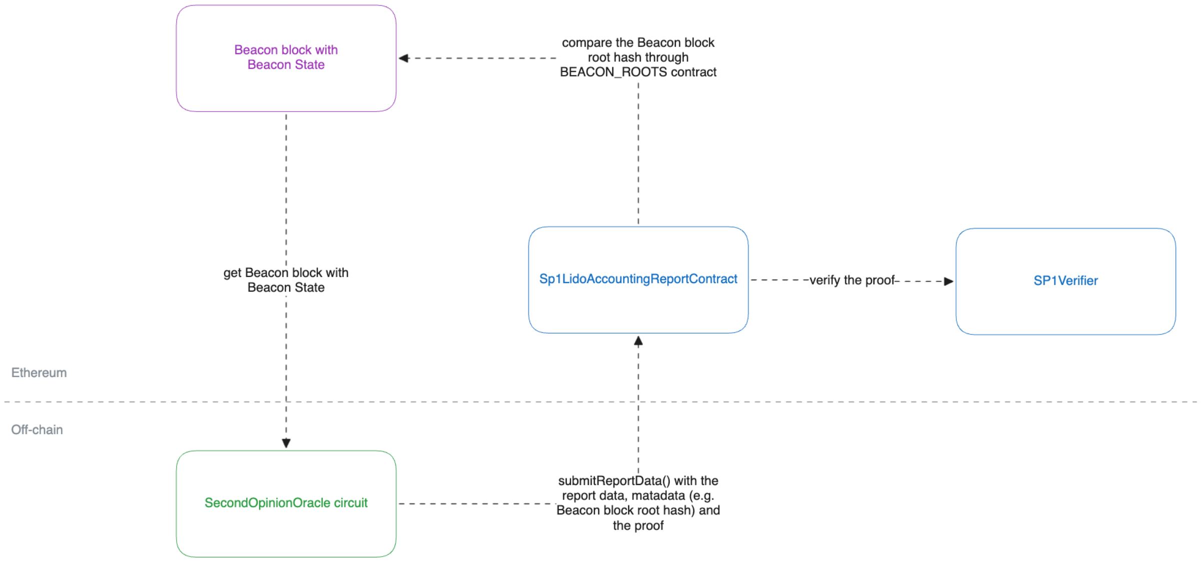

Beacon block with Beacon State
compare the Beacon block root hash through BEACON_ROOTS contract
get Beacon block with Beacon State
Sp1LidoAccountingReportContract
verify the proof
SP1Verifier
Ethereum
Off-chain
SecondOpinionOracle circuit
submitReportData() with the report data, matadata (e.g. Beacon block root hash) and the proof

Fig. 2 General Flow Diagram

4.1 Contract

The main contract is called the `Sp1LidoAccountingReportContract` and is used to submit and access proofs of past Lido states. The flow is shown in Figure 3.

The reports provided by the existing oracles are keyed by a Lido reference slot. This slot is almost equivalent to the beacon chain slot, except in the case of missed blocks. Similarly, the proofs submitted are also keyed by reference slot.

To verify the correctness of the proofs, the contract uses existing Lido state and beacon chain information. Proofs are submitted together with the public values (described in the next section), which contain the reference slot and the beacon chain slot, as well as the report. These values are validated by asserting that they correspond to the same slot and, due to EIP-4788, to the same beacon hash. Then, the SP1 plonk verifier validates the proof with the public values, and the proof is stored in a map of the reference slot to the Report. The contract also allows for an external call to retrieve the report for a provided reference slot.

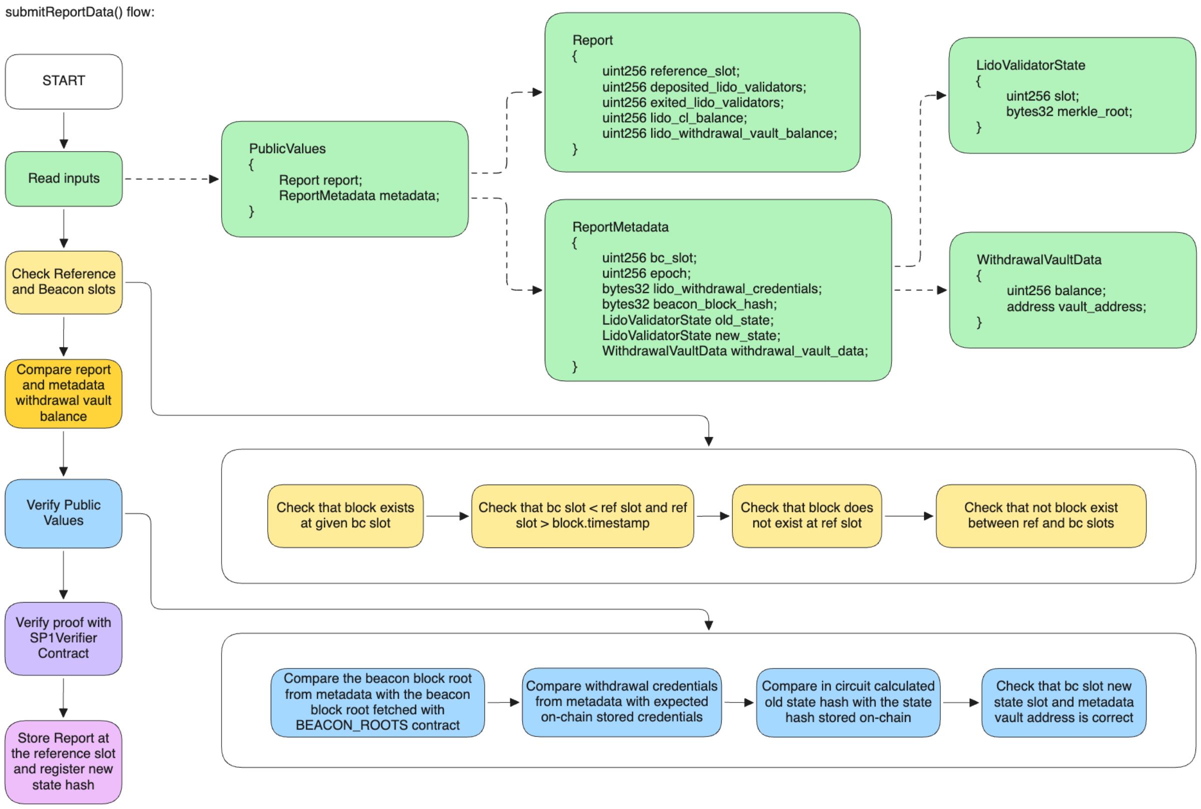
Additionally, the contract includes Pausable and Resumable functionality, allowing the contract owner to freeze the submission of new proofs for a given time period and later resume operations. During the audit, there were some changes in the contract to delegate these permissions to specific roles. The updated version implementing this change is available at commit `06bb68acdc`.

4.2 Circuit

4.2.1 SP1 Summary

zkVM's enable developers to seamlessly transition from developing unverifiable code to developing verifiable code. zkVM's follow a standard approach to generating this proof. Firstly, the code is compiled down into the target architecture and expressed in ELF format. This is then fed into a VM executor which will execute the instructions present in the ELF as well as taking snapshots of the VM's architecture, for example memory and CPU registers, enabling the prover to generate a ZK proof of each cycle. This proof can then be verified using a verification key.

SP1 is a zkVM that allows the proving of arbitrary code by compiling it into the RISC-V architecture. Most commonly this enables Rust code to be proven in ZK, but any language that can be compiled to RISC-V can be used.



submitReportData() flow:
START
Read inputs
Check Reference and Beacon slots
Compare report and metadata withdrawal vault balance
Verify Public Values
Verify proof with SP1Verifier Contract
Store Report at the reference slot and register new state hash
PublicValues { Report report; ReportMetadata metadata; }
Report { uint256 reference_slot; uint256 deposited_lido_validators; uint256 exited_lido_validators; uint256 lido_cl_balance; uint256 lido_withdrawal_vault_balance; }
ReportMetadata { uint256 bc_slot; uint256 epoch; bytes32 lido_withdrawal_credentials; bytes32 beacon_block_hash; LidoValidatorState old_state; LidoValidatorState new_state; WithdrawalVaultData withdrawal_vault_data; }
LidoValidatorState { uint256 slot; bytes32 merkle_root; }
WithdrawalVaultData { uint256 balance; address vault_address; }
Check that block exists at given bc slot
Check that bc slot < ref slot and ref slot > block.timestamp
Check that block does not exist at ref slot
Check that not block exist between ref and bc slots
Compare the beacon block root from metadata with the beacon block root fetched with BEACON_ROOTS contract
Compare withdrawal credentials from metadata with expected on-chain stored credentials
Compare in circuit calculated old state hash with the state hash stored on-chain
Check that bc slot new state slot and metadata vault address is correct

Fig. 3 On-chain Report Publishing Flow Diagram

4.2.2 Lido Oracle Circuit

The circuit is defined in the SP1 program and aims to perform validation of the oracle report.

The public values of the circuit are the report itself and the report metadata.

```
struct Report {
    uint256 reference_slot;
    uint256 deposited_lido_validators;
    uint256 exited_lido_validators;
    uint256 lido_cl_valance;
    uint256 lido_withdrawal_vault_balance;
}

struct ReportMetadataSolidity {
    uint256 bc_slot;
    uint256 epoch;
    bytes32 lido_withdrawal_credentials;
    bytes32 beacon_block_hash;
    LidoValidatorStateSolidity state_for_previous_report;
    LidoValidatorStateSolidity new_state;
    LidoWithdrawalVaultDataSolidity withdrawal_vault_data;
}

struct LidoValidatorStateSolidity {
    uint256 slot;
    bytes32 merkle_root;
}

struct LidoWithdrawalVaultDataSolidity {
    uint256 balance;
    address vault_address;
}
```

The flow is visualised in Figure 4 and performs the following:

First the program verifies the program input by checking:

a) Proving the correct beacon state by providing the correct beacon hash.

b) Proving the validity of the old lido state by checking the inputted state with the old state hash.

c) Proving the changed validators and balances exist within the beacon state.

d) Verifying the withdrawal vault input.

Next, the circuit computes the new lido validator state by merging the old state with the changes to the validators. The new Lido state has its root computed and compared with the inputted new Lido state root, if they are not the same the code panics and the proof is not generated. Finally, the public values are computed based on the values inputted and computed along the way. They are encoded and then committed.

Therefore, by verifying the state of the beacon chain and Lido states the circuit is able to generate a report for the total lido validator balances.

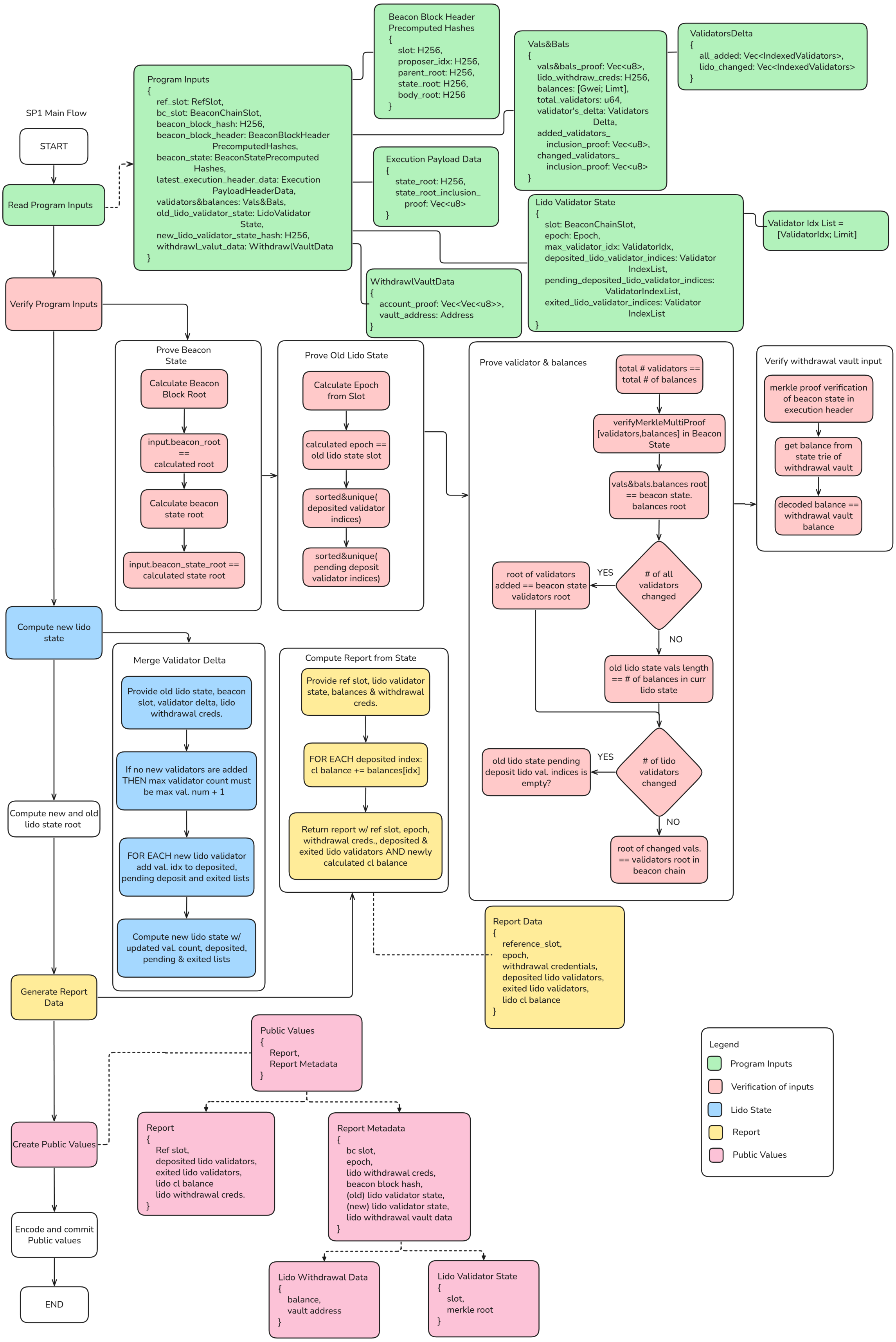

SP1 Main Flow
START
Read Program Inputs
Verify Program Inputs
Compute new lido state
Compute new and old lido state root
Generate Report Data
Create Public Values
Encode and commit Public values
END
Program Inputs { ref_slot: RefSlot, bc_slot: BeaconChainSlot, beacon_block_hash: H256, beacon_block_header: BeaconBlockHeader PrecomputedHashes, beacon_state: BeaconStatePrecomputed Hashes, latest_execution_header_data: Execution PayloadHeaderData, validators&balances: Vals&Bals, old_lido_validator_state: LidoValidator State, new_lido_validator_state_hash: H256, withdrawl_valut_data: WithdrawlVaultData }
Beacon Block Header Precomputed Hashes { slot: H256, proposer_idx: H256, parent_root: H256, state_root: H256, body_root: H256 }
Vals&Bals { vals&bals_proof: Vec<u8>, lido_withdraw_creds: H256, balances: [Gwei; Limt], total_validators: u64, validator's_delta: Validators Delta, added_validators_ inclusion_proof: Vec<u8>, changed_validators_ inclusion_proof: Vec<u8> }
ValidatorsDelta { all_added: Vec<IndexedValidators>, lido_changed: Vec<IndexedValidators> }
Execution Payload Data { state_root: H256, state_root_inclusion_ proof: Vec<u8> }
Lido Validator State { slot: BeaconChainSlot, epoch: Epoch, max_validator_idx: ValidatorIdx, deposited_lido_validator_indices: Validator IndexList, pending_deposited_lido_validator_indices: ValidatorIndexList, exited_lido_validator_indices: Validator IndexList }
Validator Idx List = [ValidatorIdx; Limit]
WithdrawlVaultData { account_proof: Vec<Vec<u8>>, vault_address: Address }
Prove Beacon State
Calculate Beacon Block Root
input.beacon_root == calculated root
Calculate beacon state root
input.beacon_state_root == calculated state root
Prove Old Lido State
Calculate Epoch from Slot
calculated epoch == old lido state slot
sorted&unique(deposited validator indices)
sorted&unique(pending deposit validator indices)
Prove validator & balances
total # validators == total # of balances
verifyMerkleMultiProof [validators,balances] in Beacon State
vals&bals.balances root == beacon state. balances root
# of all validators changed
YES
root of validators added == beacon state validators root
NO
old lido state vals length == # of balances in curr lido state
# of lido validators changed
YES
old lido state pending deposit lido val. indices is empty?
NO
root of changed vals. == validators root in beacon chain
Verify withdrawal vault input
merkle proof verification of beacon state in execution header
get balance from state trie of withdrawal vault
decoded balance == withdrawal vault balance
Merge Validator Delta
Provide old lido state, beacon slot, validator delta, lido withdrawal creds.
If no new validators are added THEN max validator count must be max val. num + 1
FOR EACH new lido validator add val. idx to deposited, pending deposit and exited lists
Compute new lido state w/ updated val. count, deposited, pending & exited lists
Compute Report from State
Provide ref slot, lido validator state, balances & withdrawal creds.
FOR EACH deposited index: cl balance += balances[idx]
Return report w/ ref slot, epoch, withdrawal creds., deposited & exited lido validators AND newly calculated cl balance
Report Data { reference_slot, epoch, withdrawal credentials, deposited lido validators, exited lido validators, lido cl balance }
Public Values { Report, Report Metadata }
Report { Ref slot, deposited lido validators, exited lido validators, lido cl balance lido withdrawal creds. }
Report Metadata { bc slot, epoch, lido withdrawal creds, beacon block hash, (old) lido validator state, (new) lido validator state, lido withdrawal vault data }
Lido Withdrawal Data { balance, vault address }
Lido Validator State { slot, merkle root }
Legend
🟩 Program Inputs
🟥 Verification of inputs
🟦 Lido State
🟨 Report
🟪 Public Values

Fig. 4 Circuit Flow Diagram

5 Risk Rating Methodology

The risk rating methodology used by Nethermind Security 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 the finding is to be uncovered and exploited by an attacker. 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 exploits 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.

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

To address issues that do not fit a High/Medium/Low severity, Nethermind Security 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 and circuit development best practices;

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

5.1 ZK Audit Issue Classification

ZK protocols must adhere to the three principles of zero-knowledge: **Completeness**, **Soundness** and **Zero-Knowledge**. The table below describes the classification of issues in ZK circuits with their corresponding principles of zero-knowledge.

Property	**Description**	**Classification**
Completeness	A valid proof must always pass with an honest verifier	Over-constrained
Soundness	An invalid proof should pass only with negligible probability	Under-constrained
Zero-Knowledge	The proof must not reveal any information about the witness	Leakage

6 Issues

6.1 [High] The cl_balance manipulation and oracle DOS through incorrect validator indices in lido_changed

File(s): input_verification.rs, lido.rs

Classification: Under-constrained.

Description: The new validators in the all_added may be included also in the list of changed validators lido_changed. This could be exploited to artificially increase the reported sum of all validator balances in cl_balance. The validators in all_added represent all the new validators that were registered between the previous and current state. All validators in the lido_changed list are ensured to be in the current beacon state:

```rust
self.verify_validator_inclusion_proof(
    &format!("{vals_and_bals_prefix}.validator_inclusion_proofs.lido_changed"),
    total_validators,
    &beacon_state.validators,
    &input.validators_and_balances.validators_delta.lido_changed,
    proof,
)?;
```

However, there is no check to ensure that the lido_changed are the validators from the old state. This allows for including the validators from all_added also into lido_changed. The function merge_validator_delta(...) checks validators' status and adds them to the pending, deposited, or exited lists. From the exploit perspective, we are only interested in creating duplicates in deposited since those are used to calculate cl_balance. This can be achieved by adding from all_added to lido_changed only those validators that currently have "Deposited" status and were not registered in the previous state. The status is determined by the status(...) function:

```rust
fn status(&self, epoch: Epoch) -> ValidatorStatus {
    if epoch >= self.exit_epoch {
        ValidatorStatus::Exited
    } else if epoch >= self.activation_eligibility_epoch {
        ValidatorStatus::Deposited
    } else {
        ValidatorStatus::FutureDeposit
    }
}
```

The merge_validator_delta(...) first iterates over the all_added and gives our targeted validators status of Deposited and adds them to new_deposited:

```rust
for validator_with_index in &validator_delta.all_added {
    let validator = &validator_with_index.validator;
    if !validator.is_lido(lido_withdrawal_credentials) {
        continue;
    }
    match validator.status(epoch) {
        // @audit: our targeted validators would be added here
        ValidatorStatus::Deposited => new_deposited.push(validator_with_index.index),
        ValidatorStatus::FutureDeposit => {
            new_pending_deposit.insert(validator_with_index.index);
        }
        ValidatorStatus::Exited => {
            new_deposited.push(validator_with_index.index);
            new_exited.push(validator_with_index.index);
        }
    }
}
```

Then, the validators in the lido_changed are distributed by comparing the status at the old epoch and new epoch:

```
for validator_with_index in &validator_delta.lido_changed {
            let validator = &validator_with_index.validator;
            // It is expected that the caller will filter out non-Lido validators, but worth double-checking
            if !validator.is_lido(lido_withdrawal_credentials) {
                return Err(Error::NonLidoValidatorInDelta {
                    index: validator_with_index.index,
                    withdrawal_credentials: validator.withdrawal_credentials,
                });
            }

            let old_status = validator.status(self.epoch);
            let new_status = validator.status(epoch);

            match (&old_status, &new_status) {
                // @audit: our targeted validators did not exist in the old state, so its old status is
                 ↪  determined as FutureDeposit.
               //              They currently have Deposited status, so those validators would be match this case
                (ValidatorStatus::FutureDeposit, ValidatorStatus::Deposited) => {
                    new_deposited.push(validator_with_index.index);
                    new_pending_deposit.remove(&validator_with_index.index);
                }
                ...
            }
        }
```

Our targeted validators in the current epoch have Deposited status and didn't exist in the old status. The function `merge_validator_delta(...)` assumes that the existence of the validators was checked before, so it treats non-existing validators as FutureDeposit. Our targeted validators would match the case: `(ValidatorStatus::FutureDeposit, ValidatorStatus::Deposited)`. In this case, the validator is added to the `new_deposited` and is removed from `new_pending_deposit`, which contains all pending validators from the old state. Our targeted validators did not exist in the old state, so they won't be in `new_pending_deposit`. Removal from the HashSet results in removing the element if it exists and returning a boolean value that informs whether the element is present. The operation `new_pending_deposit.remove(validator_with_index.index)` doesn't check the return boolean value, which for our targeted validator would be `false`. The `deposited_lido_validator_indices` would contain the duplicated values. The `compute_from_state(...)` function would iterate over duplicated validators and artificially increase the sum of balances stored in `cl_balance`.

```
pub fn compute_from_state(...) -> Self {
    let mut cl_balance: u64 = 0;

    let deposited_indices = &lido_validators_state.deposited_lido_validator_indices;
    // @audit: through duplicated validator indices, the sum of balances is artificially increased
    for index in deposited_indices {
        cl_balance += balances[u64_to_usize(*index)];
    }
   ...
}
```

Another similar scenario can happen with a validator that is both in `all_added` and `lido_changed`, but it currently has an Exited status. In the `lido_changed` loop, it would match the case `(ValidatorStatus::FutureDeposit, ValidatorStatus::Exited)`. The effects would be similar to those described above, with an additional presence in the `new_exited`. This scenario, however, is less likely since the validator can be exited 9 days from being activated while the oracle should be run daily. Additionally, the duplicated balance would exist only around 1 day while it's in the exit queue waiting for withdrawal. After withdrawal, the balance duplication is not harmful since it's zero.

Apart from the artificially increased sum of balance, in both scenarios, the Lido state hash that is registered on the chain would be incorrect. This would cause the DOS since the next fair oracle would not be able to publish correct data because the old state hash from the `Sp1LidoAccountingReportContract` would not match. Submitting the old state that would match the on-chain state would be impossible since the deposited list contains duplicates for which the circuit throws an error. Therefore, the oracle is completely blocked without the possibility of continuation.

Recommendation(s): Consider ensuring that the validators in the `lido_changed` are not new validators but were registered in the old state. This could be done by adding a check that the last index in `lido_changed` is less or equal to the `max_validator_index`. Additionally, consider checking that the boolean value of `new_pending_deposit.remove(...)` is equal to true to ensure that it exists before removing it.

Status: Fixed

Update from the client: Criticality of this issue could be discussed. According to LIP-23, the use case of ZK Oracle is only for the negative rebase. This means that increasing CL Balance during attack is useless to confirm the negative rebase. However, in general this issue is valid and fixed. Fixed in PR #12.

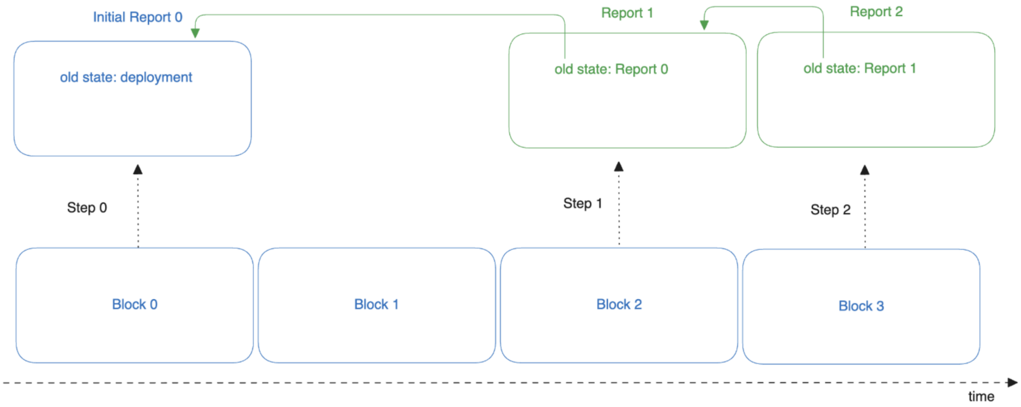
Update from Nethermind Security : After discussing with the Lido team to clarify the severity classification, they acknowledged its relevance in the context of zkOracle as a standalone project. Thus, the severity has been retained.

6.2 [Medium] Arbitrary old state can be used

File(s): `contracts/src/Sp1LidoAccountingReportContract.sol`, `crates/program/src/bin/main.rs`

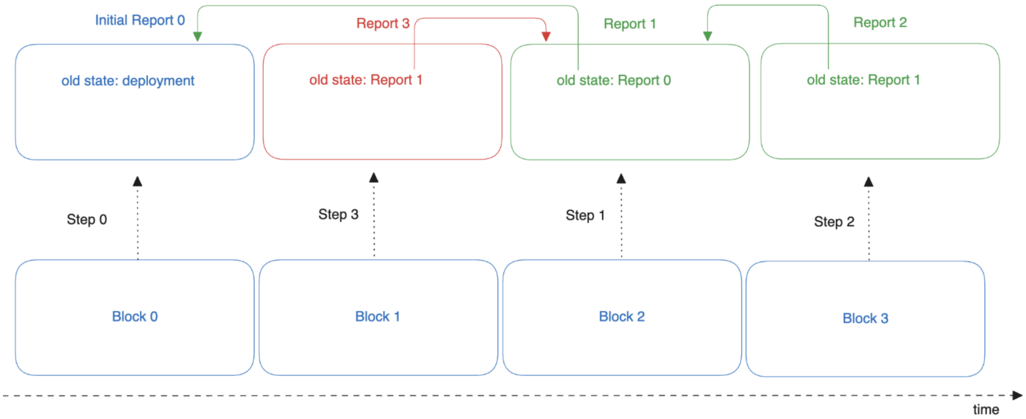
Classification: Under-constrained.

Description: The report publication includes `Report` and `ReportMetadata`. Part of the `ReportMetadata` is the `old_state` which is the previous state to which the changes are compared. Expected scenario is that the `old_state` is the root hash of the state from the last published report.

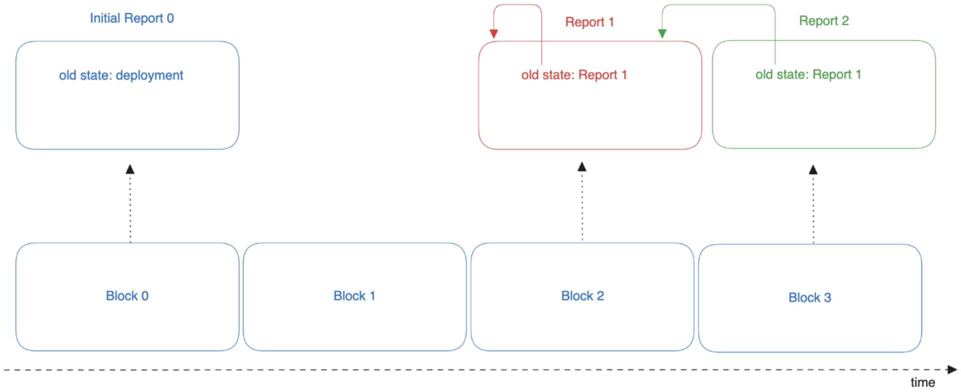


However, there are no checks to ensure that. Moreover, the reports can be published in the past, or already published reports can be republished. This allows for many scenarios that would cause incorrect state on-chain that are confusing and not expected:

- the old state can be the future state: let's assume that there are reports published at `x` and `x+2` slots. The report can be published at the `x+1` slot with `old_state` pointing to the `x+2`. This is incorrect since the future state can't be an old state. Note that for this to happen between `x+1` and `x+2` states, there can't be any changes in new validators;



- the old state can be current state: let's assume that there is report published at `x` slot with old state at `x-1`. The report can be republished at `x` with old state at `x`. The old state should not be the current state since that does not prove the correctness of provided data;



The old state should be based on which the oracle verifies the correctness of changes and shows the history of state change. In both scenarios, the old state does not provide any information. The second scenario is the most dangerous since every report can be republished this way and show `old_state` as its own state. Note that this does not allow for publishing incorrect data, however the on-chain state reports could not be trusted since they would show that the old state is equal to the current state, which does not prove correct verified change.

Recommendation(s): Consider adding a requirement in the circuit program that the new state epoch is greater than the old state epoch.

Status: Fixed

Update from the client: Fixed by adding check to the contract. PR #14

6.3 [Low] No checks on new state

File(s): `main.rs`

Classification: Under-constrained.

Description: The proof is to ensure that the new state is correctly constructed (from the old state). Currently this would potentially allow a new invalid state with duplicates to have a proof constructed.

Recommendation(s): Check that the new state is correctly computed by performing the sorted and unique checks on the new state and also checking the hash tree root of the new state is equivalent to the one passed in.

Specifically, the new state must be verified. In `main.rs`, after computing the new lido state we propose to perform the following check:

```
if !Self::is_sorted_and_unique(&mut new_state.deposited_lido_validator_indices.iter()) {
      return Err(Error::ConditionCheck(
           ConditionCheckFailure::DepositedValidatorsNotSorted,
      ));
}
// no need to check pending since it is a hash set which would not allow duplicates and also gets sorted once it is converted into a list.
```

Finally, the code must check that the computed new state root equals the input new state root. This is correctly performed in the code.

Status: Fixed

Update from the client: Accepted and fixed. Fixed at PR #17.

6.4 [Info] Inefficient checks for old lido state

File(s): main.rs

Description: To verify the validity of the old lido state passed as the private input the following checks are performed in input_verification.rs:

1. Check correct epoch value;
2. Check if the deposited validator indices are unique and sorted;
3. Check if the pending deposited validator indices are unique and sorted;

This is performed as follows in the code:

```rust
pub fn prove_input(&self, input: &ProgramInput) -> Result<(), Error> {
        ...
        self.cycle_tracker.start_span("prove_input.old_state");
                let epoch_from_slot = input.old_lido_validator_state.slot.epoch();
        let actual_epoch = input.old_lido_validator_state.epoch;
        if actual_epoch != epoch_from_slot {
            return Err(Error::ConditionCheck(ConditionCheckFailure::OldStateEpochMismatch {
                epoch: actual_epoch,
                epoch_from_slot,
            }));
        }
        self.cycle_tracker.start_span("prove_input.old_state.deposited.sorted");
        if !Self::is_sorted_and_unique(&mut
         ↪  input.old_lido_validator_state.deposited_lido_validator_indices.iter()) {
            return Err(Error::ConditionCheck(
                ConditionCheckFailure::DepositedValidatorsNotSorted,
            ));
        }
        self.cycle_tracker.end_span("prove_input.old_state.deposited.sorted");

        self.cycle_tracker.start_span("prove_input.old_state.pending.sorted");
        if !Self::is_sorted_and_unique(
            &mut input
                .old_lido_validator_state
                .pending_deposit_lido_validator_indices
                .iter(),
        ) {
            return Err(Error::ConditionCheck(
                ConditionCheckFailure::PendingDepositValidatorsNotSorted,
            ));
        }
        self.cycle_tracker.end_span("prove_input.old_state.pending.sorted");
        self.cycle_tracker.end_span("prove_input.old_state");
        ...
}
```

This requires looping through all the indices of two lists and can be expensive depending on the size of lists. Since the old state is trusted these checks do not need to occur for this value.

Recommendation(s): Since we trust the old state is correctly computed, these checks are redundant. We recommend to pass the old state root as an input in order to replace the checks described above for simplicity and efficiency. We can pass the old state hash in the inputs and do the check they are equal, similarly to the new state hash, and then finally add in the report the calculated value. Specifically, these checks above can be avoided by adding as an additional value in the ProgramInput struct which is the old lido state's hash tree root:

```rust
#[derive(PartialEq, Debug, Clone, Serialize, Deserialize)]
pub struct ProgramInput {
    ...
    pub validators_and_balances: ValsAndBals,
    pub old_lido_validator_state: LidoValidatorState,
    pub old_lido_validator_state_root: Hash256, // added this
    pub new_lido_validator_state_hash: Hash256,
    pub withdrawal_vault_data: WithdrawalVaultData,
}

```

Remove the above checks in input_verification.rs, and in main.rs add the following check:

```rust
input.old_lido_validator_state.tree_hash_root() == input.old_lido_validator_state_root
```

Why this works? As we trust the first hash of the lido state we can thus trust the previous root is correct. Additionally, merkelization results in a different root if validator indices are in incorrect order and/or are not unique so this implicitly checks the sorted and uniqueness of the indices. This provides stronger guarantees of correctness.

Status: Acknowledged

Update from the client: This optimizes the amount of data and checks to be done in the proof - at a cost of capturing issues much later in the process (at contract). Since this is an info, we would like to keep the checks in place for (1) added security, (2) added convenience in testing (tests failing proof don't need to hit the prover network).

6.5 [Info] `TimestampOutOfRange` error reports parameters in the wrong order

File(s): `Sp1LidoAccountingReportContract.sol`

Description: The `TimestampOutOfRange` error is declared as:

```
error TimestampOutOfRange(uint256 target_slot, uint256 target_timestamp, uint256 earliest_available_timestamp);
```

However, in the `_findBeaconBlockHash` function, the earliest and target timestamps are swapped. This can:

```
function _findBeaconBlockHash(uint256 slot) internal view virtual returns (bytes32) {
    uint256 targetBlockTimestamp = _slotToTimestamp(slot + 1);

    uint256 earliestBlockTimestamp = block.timestamp - (BEACON_ROOTS_HISTORY_BUFFER_LENGTH * SECONDS_PER_SLOT);
    if (targetBlockTimestamp <= earliestBlockTimestamp) {
        // @audit the timestamps order are mismatched
        revert TimestampOutOfRange(slot, earliestBlockTimestamp, targetBlockTimestamp);
    }
    ...
}
```

This could produce error data that leads to misinterpretation of error messages.

Recommendation(s): Consider applying the following change:

```
-revert TimestampOutOfRange(slot, earliestBlockTimestamp, targetBlockTimestamp);
+revert TimestampOutOfRange(slot, targetBlockTimestamp, earliestBlockTimestamp);
```

Status: Fixed

Update from the client: Fixed a61db08

6.6 [Info] Redundant merkle proof check

File(s): `input_verification.rs`

Description: There is a merkle multiproof check that verifies that the validator hash and the balances hash from the inputted beacon state. This can be seen below:

```
let vals_and_bals_multiproof_leaves = [beacon_state.validators, beacon_state.balances];
beacon_state
    .verify_serialized(
        &input.validators_and_balances.validators_and_balances_proof,
        &[BeaconStateFields::validators, BeaconStateFields::balances],
        &vals_and_bals_multiproof_leaves,
    )
    .map_err(|e| Error::MerkleProofError {
        operation: "Verify Inclusion Proof for validators and balances",
        error: e,
    })?;
```

As can be seen in the code, the balances and validators hashes are already taken from the beacon state itself which makes proving their inclusion in their own struct redundant.

Recommendation(s): We recommend to remove this proof from the inputs and not perform this check.

Why can we remove this check since we need to prove the validators and balances changed and added are included in the beacon state? This will not break the chain of proofs leading from validators/balances to beacon root since those hashes in the beacon state are used to verify the inclusion of the validators and balances in question and since the beacon header and root are checked against the public beacon root. This means that we know the validators are included in the validators list which is included in the beacon state which is included in the beacon root (equivalently for the balances).

Existing check for the added validators inclusion proof that proves validators $\in$ validator list:

```
let proof =
    merkle_proof::serde::deserialize_proof(&input.validators_and_balances.added_validators_inclusion_proof)
        .map_err(|err| Error::MerkleProofError {
            operation: "Deserialize all_added Inclusion Proof",
            error: err,
        })?;
self.verify_validator_inclusion_proof(
    &format!("{vals_and_bals_prefix}.validator_inclusion_proofs.all_added"),
    total_validators,
    &beacon_state.validators,
    &input.validators_and_balances.validators_delta.all_added,
    proof,
```

Existing check that beacon state $\in$ beacon root:

```
let beacon_block_header = &input.beacon_block_header;
let beacon_state = &input.beacon_state;

// Beacon Block root == merkle_tree_root(BeaconBlockHeader)
self.cycle_tracker.start_span("prove_input.beacon_block_header");
let bh_root = beacon_block_header.tree_hash_root();
if bh_root != input.beacon_block_hash {
    return Err(Error::ConditionCheck(ConditionCheckFailure::BeaconBlockHashMismatch {
        actual: bh_root,
        expected: input.beacon_block_hash,
    }));
}
self.cycle_tracker.end_span("prove_input.beacon_block_header");

// merkle_tree_root(BeaconState) is included into BeaconBlockHeader
self.cycle_tracker.start_span("prove_input.beacon_state");
let bs_root = beacon_state.tree_hash_root();
if bs_root != beacon_block_header.state_root {
    return Err(Error::ConditionCheck(ConditionCheckFailure::BeaconStateHashMismatch {
        actual: bs_root,
        expected: beacon_block_header.state_root,
    }));
}
self.cycle_tracker.end_span("prove_input.beacon_state");
```

On the other hand, should this check be kept we recommend to add the `latest_execution_payload_header` to the values proven for completeness since it is also used later in the circuit.

Status: Fixed

Update from the client: Fixed. PR #17

6.7 [Info] Unnecessary data in `lido_changed`

File(s): `lido.rs`

Description: The data preparation script in `validator_delta.rs::compute_changed(...)` gathers the validators to `lido_changed`. All the pending validators from the old state are added. Then, it iterates over the deposited validators in the old state and checks if the `activation_epoch` has changed:

```
for index in &self.old_state.deposited_lido_validator_indices {
    // for already deposited validators, we want to check if something material has changed:
    // this can only be an activation epoch or exist epoch. Theoretically "slashed" can also be
    // relevant, but for now we have no use for it
    let index_usize = u64_to_usize(*index);
    let old_validator = &self.old_bs.validators[index_usize];
    let new_validator = &self.new_bs.validators[index_usize];

    if !self.skip_verification && old_validator.pubkey != new_validator.pubkey {
        return Err(Error::ValidatorPubkeyMismatch {
            index: *index,
            old: old_validator.pubkey.clone(),
            new: new_validator.pubkey.clone(),
        });
    }
    if check_epoch_based_change(
        old_bs_epoch,
        new_bs_epoch,
        old_validator.exit_epoch,
        new_validator.exit_epoch,
    ) {
        lido_changed_indices.insert(*index);
    }
    if check_epoch_based_change(
        old_bs_epoch,
        new_bs_epoch,
        old_validator.activation_epoch,
        new_validator.activation_epoch,
    ) {
        lido_changed_indices.insert(*index);
    }
}
```

In case the `activation_epoch` is changed for a validator, then it is included in the `lido_changed`. The circuit handles `lido_changed` in the `lido.rs::merge_validator_delta(...)`. In the presented scenario, our validator previously had Deposited since it was in the deposited validators. The `merge_validator_delta(...)` would check its status, and the new status would also be Deposited. It would match the `(ValidatorStatus::Deposited, ValidatorStatus::Deposited)` case, which would result in no operation.

Recommendation(s): Consider not including the validators that had Deposited status in the old state to the `lido_changed` if their activation epoch changed. This would save computation and proving time.

Status: Fixed

Update from the client: Fixed. PR #17

6.8 [Info] Update crates/program rust-toolchain

File(s): `crates/program/rust-toolchain`

Description: Originally, the `channel` had to be 1.82 because of SP1. As specified in the code below:

```
> ==== cargo edition and rustc compatibility ====
>  These dependencies are pinned to a concrete version as a compatibility mechanism between new version of the crates
>  (moving on oo edition=2024 and rust-version=1.85) and sp1 compiler that's based on rustc=1.82
>  This can be unpinned when sp1 rustc moves to at least 1.85
>  1.2.0 requires rust >= 1.85, sp1 uses 1.82-dev
>  Note: there are overrides in script and service Cargo.toml in dev-dependencies - needs to be updated there as well
```

sp1 rustc has moved to at least 1.85, so the value can be updated to maintain consistency, reduce maintenance overheads and access the bug fixes included in 1.85

Recommendation(s): Change `channel = "1.82"` to `channel = "1.85"`.

Status: Fixed

Update from the client: Fixed. PR #11

6.9 [Best Practices] Unchecked integer overflows

File(s): `crates/shared/src/lib/circuit_logic/report.rs`, crates/shared/src/lib/circuit_logic/inpt_verification.rs

Description: Some arithmetic operations in the codebase are performed without explicit overflow checks. Since SP1 operates over release mode, overflow behavior does not panic no immediate security vulnerability has been identified, as overflows would require unrealistically large inputs. However, the absence of explicit checks could lead to unexpected behavior if malicious input is used.

Recommendation(s): Consider integrating checks like `checked_add()` to the calculations e.g. of the accumulated balance and total validators. Using `overflow-checks = true` on `Cargo.toml` won't be enough on the SP1 platform.

Status: Fixed

Update from the client: Fixed. PR #16 and PR #22

7 Documentation Evaluation

Software documentation refers to the written or visual information describing software's functionality, architecture, design, and implementation. 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 is critical in software development, enabling effective communication between developers, testers, users, and other stakeholders. 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. Protocols can use various types of software documentation for their circuits. Some of the most common types include:

Circuit Specifications: A documentation that outlines the purpose, functionality, and constraints of the circuit:

i) High-level description - What the circuit is designed to achieve, e.g., proving membership in a Merkle tree, verifying a signature, etc. Description about inputs and outputs (public and private signals);

ii Mathematical model - Description of the mathematical computations and proofs being implemented. In addition, describe the cryptographic primitives that are used, e.g., Poseidon hash, EdDSA verification.

iii Expected behavior - Present a detailed explanation of how the circuit behaves under normal, edge, and adversarial conditions.

Documentation of the Input and Output: The documentation should describe details of the inputs and outputs of the circuit.

i) Public inputs: List of public inputs and their expected values and ranges. Also, clarify constraints on the public inputs (e.g., nonzero).

ii) Private inputs: List of private input signals and their roles in the circuit. The documentation should also include validation requirements for private inputs within the circuit.

iii) Outputs: A description of the outputs, their intended use and their relationship with the inputs.

Performance Metrics: Describe details on the prover and verifier performance such as, time taken to generate proofs, memory usage during proof generation, time taken to verify proofs, etc.

Testing Documentation: Testing documentation is a document that provides information about how the circuits were tested. It includes details on the test cases that were used, coverage metrics showing which parts of the circuit have been tested, and any issues that were identified during testing.

Technical whitepaper: A technical whitepaper is a comprehensive document that describes the design and technical details of the protocol. It includes information on the purpose of the protocol, contracts, its architecture, its components, and how they interact with each other;

Summary: These types of documentation are essential for circuit development and maintenance. They help ensure that the circuit is properly designed, implemented, and tested. The documentation provides a reference for developers who need to modify the circuit in the future.

Remarks about the Lido Accounting ZK Oracle built on Succint SP1 documentation

The **Lido Accounting ZK Oracle** protocol documentation was made available through lip-23 and oracle-report-sanity-checker with a high-level overview. In addition, the team provided audit-specific documentation, detailing the core contracts and files, outlining the flow step by step with comments in the code. Furthermore, the **Lido** team thoroughly addressed all questions and concerns raised by the **Nethermind Security** team, providing valuable insights and a comprehensive understanding of the project's technical aspects.

8 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.

A Appendix: Deployment Scripts

This section presents some points of attention identified during the review of the deployment script. We would like to highlight that the issues A.2 and A.3 are outside the scope of this audit. They were found during our code review to understand the deployment, debugging process, etc. Therefore, we cannot confirm the absence of further issues, as these files were not fully audited.

A.1 Ambiguous Placeholder Usage for API Keys in Environment Configuration

File(s): `.env.example`

Description: The current `.env.example` file uses a single placeholder (`$$API_KEY$$`) for multiple RPC endpoints, including both the Execution Layer (Infura) and Consensus Layer (Ankr). As each service relies on a different infrastructure provider, they require distinct API keys. Using a shared placeholder introduces ambiguity and often requires manual intervention to correct. This issue also affects other environment variables such as `FORK_URL` and `INTEGRATION_TEST_FORK`, which currently rely on the same generic placeholder.

Current Configuration

```
EXECUTION_LAYER_RPC=https://mainnet.infura.io/v3/$$API_KEY$$
CONSENSUS_LAYER_RPC=https://rpc.ankr.com/premium-http/eth_beacon/$$API_KEY$$
FORK_URL=https://mainnet.infura.io/v3/$$API_KEY$$
INTEGRATION_TEST_FORK_URL=https://rpc.ankr.com/premium-http/eth_sepolia_beacon/$$API_KEY$$
```

Potential for runtime failure **If `$$API_KEY$$`** is replaced with only one key, at least one of the services will fail due to an invalid key.

Recommendation(s): Update the `.env.example` file to use distinct, clearly named placeholders for each service:

```
EXECUTION_LAYER_RPC=https://mainnet.infura.io/v3/$$INFURA_API_KEY$$
CONSENSUS_LAYER_RPC=https://rpc.ankr.com/premium-http/eth_beacon/$$ANKR_API_KEY$$
FORK_URL=https://mainnet.infura.io/v3/$$INFURA_API_KEY$$
INTEGRATION_TEST_FORK_URL=https://rpc.ankr.com/premium-http/eth_sepolia_beacon/$$ANKR_API_KEY$$
```

Status: Fixed

Update from the client: Fixed. `PR #16`

A.2 Inconsistency: Sensitivity Flag for Execution vs. Consensus Layer RPC

File(s): `crates/script/src/lib/env.rs`

Description: The environment variable configuration flags `EXECUTION_LAYER_RPC` as **non-sensitive**, while `CONSENSUS_LAYER_RPC` is marked as **sensitive**, as presented below:

```
pub const EXECUTION_LAYER_RPC: EnvVarSpec = EnvVarSpec {
    key: "EXECUTION_LAYER_RPC",
    sensitive: false,
};

pub const CONSENSUS_LAYER_RPC: EnvVarSpec = EnvVarSpec {
    key: "CONSENSUS_LAYER_RPC",
    sensitive: true,
};
```

This leads to potential exposure of the execution layer API key in logs:

①. *Both values typically contain API keys*. Since both variables embed URLs with provider-specific API keys (for instance, `https://rpc.ankr.com/.../<key>` and `https://mainnet.infura.io/v3/<key>`), **both should be considered sensitive**. These keys are credentials granting access to infrastructure services.

②. *Inconsistent redaction in logs*. The `Debug` implementation for `EnvVarValue<T>` uses this `sensitive` flag to redact values in logs:

```
let value_print = if self.spec.sensitive {
    "***".to_string()
} else {
    format!("{:?}", self.value)
};
```

Then:

Ⓐ. The **Consensus Layer RPC** will be redacted in logs.
Ⓑ. The **Execution Layer RPC** will be printed in full, exposing the API key.

Recommendation(s): For consistency and security, both should be marked `sensitive: true`:

```rust
pub const EXECUTION_LAYER_RPC: EnvVarSpec = EnvVarSpec {
    key: "EXECUTION_LAYER_RPC",
    sensitive: true, // changed from false
};
```

Status: Fixed

Update from the client: Fixed. PR #16

A.3 Sensitive values are manually logged, ignoring the `sensitive` flag

File(s): `crates/script/src/lib/scripts/prelude.rs`

Description: In the `EnvVars::for_logging` function, `.value.to_string()` calls directly log **the full raw values**, regardless of whether the variable is marked as `sensitive` in its `EnvVarSpec`.

```rust
if !only_important {
    ...
    result.insert("execution_layer_rpc", self.execution_layer_rpc.value.to_string());
    result.insert("consensus_layer_rpc", self.consensus_layer_rpc.value.to_string());
    ...
}
```

`EnvVarValue<T>` already implements a secure `Debug` trait that respects `sensitive`:

```rust
#[derive(Clone, Copy)]
pub struct EnvVarSpec {
    pub key: &'static str,
    pub sensitive: bool,
}

#[derive(Clone, Copy)]
pub struct EnvVarValue<TVal> {
    pub spec: &'static EnvVarSpec,
    pub value: TVal,
}
```

However, `EnvVars::for_logging` bypasses this safety step and manually converts `.value` to string — this may **reveal API KEYS** for any mistakenly marked variables. For example:

`consensus_layer_rpc`: correctly marked as `sensitive: true` — **but still printed**.

Recommendation(s): Instead of manually calling `.value.to_string()`, use the `Debug` implementation of `EnvVarValue<T>` to automatically redact sensitive values. This ensures that sensitive values print as "***" and non-sensitive values print their actual content — as designed by:

```rust
impl<TVal: Debug> Debug for EnvVarValue<TVal> {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        let value_print = if self.spec.sensitive {
            "***".to_string()
        } else {
            format!("{:?}", self.value)
        };
        ...
    }
}
```

Status: Fixed

Update from the client: Fixed. PR #16

A.4 Current deploy command does not verify the deployed contract

File(s): `justfile`

Description: The current setting `verify_contract:="false"` in `justfile` does not verify the deployed contract on Etherscan.

Recommendation(s): Consider setting the `verify_contract` to `"true"` to verify the contract.

Status: Acknowledged

Update from the client: This only causes minor inconvenience during contract deployment. Fix is contained to a “development script” so won’t affect anything. This will not be fixed for now.