OpenZeppelin | security

Compounding Staking Strategy Audit

ORIGIN

September 22, 2025

Table of Contents

Table of Contents 2

Summary 4

Scope 5

Important Note on the Fix Review 5

System Overview 6

Pectra Upgrade 6

Integration with the OETH Vault 6

Proving Consensus Layer State 7

Validator Management on the Consensus Layer 7

Strategy Validator Lifecycle and Beacon Chain Interactions 8

Calculation of the Total Balance of the Strategy 9

Security Model and Trust Assumptions 10

Privileged Roles 11

Critical Severity 12

C-01 Postponed Pending Deposit Breaks the Strategy 12

High Severity 14

H-01 Deposits With Duplicate depositDataRoot Overwrite Accounting 14

H-02 Validator Registration Frontrunning Can DoS the Strategy 15

Medium Severity 16

M-01 Deposits to Exiting Validators Can Cause Undercounting 16

M-02 Leaked or Malicious registrator Can Damage the Strategy 18

Low Severity 20

L-01 Variable Names Too Similar 20

L-02 verifyBalances Can Revert for a Pending Deposit 20

L-03 The ValidatorWithdraw Event Can Emit Misleading Values 21

L-04 The stakeEth Function Does Not Leverage ETH That Exists in the Contract 21

L-05 Lack of Input Validations 22

Notes & Additional Information 23

N-01 Valid Single-Leaf Proofs Are Always Rejected 23

N-02 Missing Named Parameters in Mappings 23

N-03 Multiple Function Do Not Respect the Checks Effects Interactions (CEI) Pattern 24

N-04 Unused Code 24

N-05 Lack of Security Contact 25

N-06 _disableInitializers() Not Being Called from Initializable Contract Constructors 25
N-07 Code Simplifications 26
N-08 Missing, Incorrect, or Misleading Documentation 27

Recommendations 29

Design Considerations 29
Monitoring Recommendations 29

Conclusion 30

Summary

Type	DeFi
Timeline	From 2025-08-06 To 2025-08-20
Languages	Solidity

Total Issues	18 (10 resolved, 2 partially resolved)
Critical Severity Issues	1 (0 resolved, 1 partially resolved)
High Severity Issues	2 (2 resolved)
Medium Severity Issues	2 (0 resolved, 1 partially resolved)
Low Severity Issues	5 (3 resolved)
Notes & Additional Information	8 (5 resolved)

Scope

OpenZeppelin audited pull request #2559 of the OriginProtocol/origin-dollar repository, including all the changes made up to commit 346ca88.

In scope were the following files:

```
contracts
└── contracts
    ├── beacon
    │   ├── BeaconProofs.sol
    │   ├── BeaconProofsLib.sol
    │   ├── BeaconRoots.sol
    │   ├── Endian.sol
    │   ├── Merkle.sol
    │   └── PartialWithdrawal.sol
    ├── interfaces
    │   └── IBeaconProofs.sol
    ├── proxies
    │   └── Proxies.sol
    └── strategies
        └── NativeStaking
            ├── CompoundingStakingSSVStrategy.sol
            ├── CompoundingValidatorManager.sol
            └── ValidatorRegistrator.sol (diff)
```

Important Note on the Fix Review

The fix review for this audit covered all changes in Solidity files between the original audit commit 346ca88 and commit 30dfacb. The changes between these commits included additional fixes and changes in response to an audit performed with another security firm.

Lastly, the fixes added the possibility of the governor or registrator to pause the strategy. While paused, validators and no longer be registered and ETH can no longer be staked. Only the governor is allowed to unpause the strategy.

System Overview

In the Origin Protocol, strategies act as yield-bearing mechanisms for the Origin Dollar (OETH) and accumulate yield which is then distributed to OETH holders. The new **Compounding Staking Strategy** is a yield-generating module for OETH, natively staking ETH to earn consensus and execution rewards. The strategy is similar to the old `NativeStakingSSVStrategy`, but it leverages new technology and EIPs introduced in the Pectra Upgrade.

Pectra Upgrade

The Pectra upgrade of the consensus layer introduces several important improvements to Ethereum's Beacon chain, including increased validator flexibility and better staking operations. The key features relevant to this audit are:

- EIP-7251, which raises the maximum effective validator balance to 2048 ETH and introduces consolidations. Note that while supported, consolidations are not yet implemented within the `CompoundingStakingSSVStrategy` and are part of future work.
- EIP-7002, which introduces partial or full withdrawals initiated on the execution layer.
- EIP-6110, which enables faster and more secure validator deposits.
- EIP-7685, which creates a general-purpose framework to expose execution-layer requests to the consensus layer.

Integration with the OETH Vault

The `CompoundingStakingSSVStrategy` implements the usual Origin strategy functions, such as:

- `deposit` and `depositAll`
- `withdraw` and `withdrawAll`
- `checkBalance`

The strategy only supports WETH for deposits and withdrawals. It is important to note that if used as a default strategy, OETH redemptions will have to be supported by the Origin vault buffer. If the buffer does not have enough funds, redeems can revert, as ETH withdrawals from the Beacon chain take time and cannot be done within the same transaction.

The strategy also has functionality to:

- prove consensus layer state through Merkle proofs against Beacon block roots
- manage validators on the consensus layer
- accurately account for the total value held in the strategy (ETH in contract, WETH in contract, ETH in pending deposits on Beacon Chain, and ETH in validator balances on the Beacon chain)

Proving Consensus Layer State

Traditionally, staking balances and rewards are tracked using an oracle, which can suffer from issues such as liveness or information staleness. The `CompoundingValidatorManager` contract utilizes the standalone `BeaconProofs` contract to verify Beacon chain state through Merkle proofs against the Beacon block state root of the previous block. The block state root is served through the Beacon block roots system address added in EIP-4788. For a particular timestamp, the system address will serve the Beacon block root of the parent block and can provide data for up to the previous 8191 blocks which provides about a day of coverage.

Validator Management on the Consensus Layer

The Origin protocol manages validators through SSV network staking which uses Distributed Validator Technology (DVT) for optimal liveness, security, and decentralization of the validators. The strategy's off-chain infrastructure utilizes the P2P API to generate private keys and key shares. Subsequently, the key shares will be leveraged to register the validators into the SSV network.

The strategy can register and remove validators from the SSV network, as well as withdraw any excess SSV tokens not used to pay for SSV network fees. Note there is no functionality to deposit SSV tokens to a particular cluster, since this can be done by any address and will be handled by the strategist.

Strategy Validator Lifecycle and Beacon Chain Interactions

The Origin Protocol keeps an internal accounting for its validators, which could be in any of the following states: `NON_REGISTERED`, `REGISTERED`, `STAKED`, `VERIFIED`, `EXITED`, and `REMOVED`.

A partially trusted `registrator` role is responsible for registering the validator on the SSV network by calling the `registerSsvValidator` function. Once the validator is `REGISTERED`, the `registrator` can call the `stakeEth` function to deposit the initial 32 ETH required for activation to the `BEACON_CHAIN_DEPOSIT_CONTRACT`. This function also updates the state of validator to `STAKED`.

The deposit is added to the pending deposit list on the consensus layer and will take a variable amount of time to be processed depending on the state of the list. Once the deposit is processed, the `verifyValidator` function is used to prove the existence of the validator's public key on the consensus layer state. This function will mark the validator as `VERIFIED`, which is a required step before proving that a deposit was processed on the consensus layer. After validator verification, the `registrator` role can also top up the validator using the `stakeEth` function with any amount.

To prove any successful processing of a deposit on the consensus layer, the `verifyDeposit` function is called, which marks the deposit as processed if the Merkle proofs are valid against the Beacon block root.

The `registrator` can also call the `validatorWithdrawal` function to initiate an execution-layer partial or full withdrawal. This request will be passed to the consensus layer through the EIP-7002 precompile, and will be processed in a variable amount of time, depending on the voluntary withdrawals pending list on the consensus layer.

Once a validator's balance reaches zero, the `CompoundingStakingSSVStrategy` contract will assume that the validator was fully exited and mark it accordingly. Subsequently, the `registrator` could remove the validator from the SSV network through the `removeSsvValidator` function. Note that the function reverts if the validator was not yet deemed `EXITED`, meaning that intentionally slashing an active validator by no longer operating it is not possible.

Proving a Deposit Was Processed on the Consensus Layer

In order to prove on the execution layer that a deposit was processed on the consensus layer, the strategy leverages the pending deposit list. The strategy assumes that the deposits inside are in strictly increasing order based on the slot at which they were initiated. Hence, a deposit initiated at slot sX will be considered processed if $sY > sX$, where sY is the slot of the first pending deposit in the list. Then, for a particular deposit X, the strategy will pick a slot at which sY is bigger than sX, and prove this state of the consensus layer through the Beacon block root. Once proven, the deposit will be removed from the strategy's pending deposits accounting.

Proving a Validator's Balance Within the Consensus Layer

The balances of the strategy's validators are verified through the `verifyBalances` function in 2 phases:

1. The `ValidatorBalancesContainer` is proved against the Beacon state root.
2. Each of the validators' balances is proved against the `ValidatorBalancesContainer` root.

The `verifyBalances` function will run against a particular Beacon state block root, which must have been previously snapshotted by the `registrator` using the `snapBalances` function each time.

Calculation of the Total Balance of the Strategy

To calculate the total balance of the strategy, multiple elements are taken into consideration:

- The current amount of WETH inside the strategy.
- The last verified ETH balance, which is computed for a particular timestamp. It is made up of ETH within pending and not yet verified deposits, ETH within verified validator balances, and the ETH balance of the contract at the time of snapshotting. The snapshotting mechanism is relevant and necessary in order to not undercount or doublecount validator balances due to the different timings at which the validator balances will be updated and visible within each of the consensus and execution layers. The mechanism is similar to EigenLayer's Checkpoint Proof System, which can be further explored here.

Security Model and Trust Assumptions

The following trust assumptions were made about the codebase during the audit:

- The deployer of the strategy contract will set the correct parameters for the `constructor`. The governor will act in the best interest of the protocol, correctly initializing the contract, setting a `registrator`, as well as withdrawing any excess SSV tokens from the SSV network contracts.
- The `strategist` is responsible for depositing SSV tokens directly on the SSV network via the `deposit` function on behalf of the strategy. This would be to ensure sufficient SSV token balance of the cluster in the SSV network at all times.
- The `strategist` is expected to act honestly and correctly add the strategy to the vault, as well as set it as the default strategy only if the vault buffers are observed to be high enough and able to consistently serve OETH redeems.
- The P2P API and SSV network operators are assumed to secure the validator's private key and not sign any message that could lead to front-running of the deposit to the `BEACON_DEPOSIT_ADDRESS`. However, they are partially trusted entities, and recommendations regarding minimizing the front-running impact have been made in this report.
- The `registrator` role is considered a partially trusted party in the sense that if it is ever compromised or behaves maliciously, the potential impact should be as limited as possible. Further recommendations on minimizing malicious `registrator` impacts were made in the M-02 issue.
- It is possible for the strategy to lose funds due to the slashing amounts exceeding the rewards. This is relevant as OETH strategies should always have a 1-1 backing.
- It is not possible to differentiate between ETH that is sent externally to the strategy and ETH that is sourced from staking redemptions and rewards.
- SSV tokens are fully approved to the SSV contracts, and these approvals cannot be revoked.

Privileged Roles

Throughout the codebase, the following privileged roles were identified:

- The `registrator` role is responsible for registering the validators on the SSV network and the Beacon chain, depositing the funds on active validators, timely verifying validator balances, and managing the withdrawal process for the validators in the strategy.
- The `governor` role is responsible for withdrawing SSV tokens from the SSV contract and updating the `registrator` role.

Critical Severity

C-01 Postponed Pending Deposit Breaks the Strategy

When ETH is staked from the execution layer to the consensus layer, it first enters the pending deposit list, which takes multiple epochs to be processed. Once processed, a deposit will increase the balance of the validator on the Beacon chain. The `CompoundingStakingSSVStrategy` strategy assumes that the pending deposit list has a **strictly increasing** order, based on the slot at which each deposit was issued. If the first pending deposit's slot on the Beacon chain is sX, then all the deposits with slots less than sX must have been processed. Based on this assumption, the contract leverages the Beacon chain's first pending deposit slot and compares it against a deposit Y's deposit slot sY in order to verify whether Y has been processed on the Beacon chain or is still pending.

In practice, the `verifyDeposit` function considers a deposit Y to be processed if the `firstPendingDepositSlot` (i.e., first pending deposit's slot in `Deposit List` on Beacon Chain) is greater than the slot at which Y was issued. Otherwise, the deposit is still pending. Similarly, the `verifyBalances` function requires all unverified deposits to still be pending, as otherwise, it reverts and requires a call to the `verifyDeposit` function. However, according to the consensus layer specification, any pending deposits in the Beacon chain on an exiting validator will be **postponed to the end** of the list for every epoch.

This invalidates the assumption of the `Pending Deposit List` being in strictly increasing order, and can lead to the following impacts:

- **Double Counting**: A scenario may occur where the `firstPendingDepositSlot` list is lower than a deposit that has been processed but not yet verified. In this case, `verifyBalances` function would incorrectly treat the deposit as unprocessed and include it in the balance calculation. Given that the Beacon chain has already updated the validator balance, the increase will be counted a second time through the validator balance proofs.

- **Under Accounting**: The `verifyDeposit` function could verify a deposit X as soon as it encounters a `firstPendingDepositSlot` list that is bigger than the slot for X. This will mark the deposit as verified, even though it has not yet been processed on the

Beacon chain, leading to undercounting in the `verifyBalances` function. Note that this scenario additionally requires that Origin's validator is exiting and is the one having a postponed deposit. Otherwise, X would indeed be processed if its slot were smaller than `firstPendingDepositSlot`.

- **Temporary Denial of Service (DoS)**: Given a deposit X that was processed on the Beacon chain, and assuming that the entire list is filled with **postponed** deposits with slots lower than the slot of X, the `verifyDeposit` function will revert until a later, non-postponed deposit reaches the beginning of the list or the whole list is processed. Otherwise, there will be no valid slot to compare X's slot against.

- **Permanent Halt**: If a validator is exited on the Beacon chain, but the validator's postponed deposit has not yet been verified, it is possible that the `verifyBalances` function updates the state of the validator to `EXITED`. Once the deposit is processed, the `verifyDeposit` function will not be able to verify due to the validator state having changed to the `EXITED` state. Thus, once that deposit is processed, the `verifyBalances` function will always revert, asking to verify the pending deposit first. Note that this scenario also requires that Origin's validator is the one exiting and having a postponed deposit.

Given that the pending deposit list is not always in strictly increasing order, consider devising another mechanism that verifies if a pending deposit has already been processed on the Beacon chain.

Update: *Partially resolved.*

> *The double counting scenario is no longer present, as the `verifyBalances` function was updated to account for both balances in pending deposits and on the consensus layer through Merkle proofs.*
>
> *For the under accounting case, the `stakeEth` function will only allow deposits to validators that are `REGISTERED`, `VERIFIED` or `ACTIVE`. A `VERIFIED` or `ACTIVE` validator will become `EXITING` upon requesting a full withdrawal, which reduces the risk of depositing to an exiting Origin validator. Note that this risk is however still present, as the validator could be exiting on the consensus layer due to slashing.*
>
> *The temporary DoS vector is still present. The Origin Protocol team has acknowledged and accepted this risk, given the scenario is unlikely to happen for a significant amount of time. Note that this also causes a DoS of the `verifyBalances` function, since a deposit that was processed but not yet verified can not be proven on the consensus layer. To verify this deposit and unblock `verifyBalances`, the Origin Protocol team*

> *would have to wait for a moment where the first pending deposit in the consensus layer list is not a postponed deposit. These conditions will take variate amounts of time to happen, depending on network activity.*
>
> *The permanent halt scenario is no longer possible, a validator is no longer marked as `EXITED` if there are still pending, unverified deposits to it.*
>
> *Lastly, it is important to note that deposit verification should be done as soon as possible, following the deposit being processed on the consensus layer. On the contrary, the following temporary DoS can appear:*
>
> - A pending deposit is processed on the consensus layer but not yet verified. In the meantime, the validator is slashed on the consensus layer and starts exiting. This will cause the `verifyDeposit` function to revert, unless the pending deposit list is empty.
> - If at least one day has passed since the deposit was processed, the deposit can no longer be verified using a past beacon block root, as older ones would no longer be available. State roots are persisted only for the most recent 8191 blocks.
> - To verify the deposit, one would have to wait until the validator has fully exited.

High Severity

H-01 Deposits With Duplicate `depositDataRoot` Overwrite Accounting

The `stakeEth` function of the `CompoundingValidatorManager` contract records a deposit and a `depositRoot` every time an execution layer deposit goes through. The key used for the `deposits` mapping is `depositDataRoot`, which is a hash of the deposit's validator `pubkey`, `withdrawal_credentials`, `amount`, and `signature`. Note that none of these fields is necessarily unique. Hence, it is possible that two duplicate `depositDataRoots` are valid deposits and will be accepted as leaves within the Beacon chain pending deposits list.

This is problematic for the `CompoundingValidatorManager` contract, as a secondary identical `depositDataRoot` will overwrite the first deposit, causing loss of information. At the very least, this will also lead to a DoS for the `verifyBalances` function, as one of the two deposits will no longer be verifiable as per the scenario given below:

1. Both deposits are processed within the Beacon chain.
2. The `verifyDeposit` function is called, which marks the status of the latest `depositDataRoot` as `VERIFIED`. Note that `verifyDeposit` can not be called again on the same `depositDataRoot`, as it requires the status to be `PENDING`.
3. The `verifyBalances` function will revert, as it notices a deposit root that was processed on the Beacon chain but was not yet verified.
4. The `verifyBalances` function will permanently revert from that point onwards. The contract cannot recover from this state and will require a significant contract upgrade.

The possibility of duplicate `depositDataRoots` is significant, given that deposits for the same amount and validator will map to the same signature and `depositDataRoot` under a deterministic ECDSA algorithm. Even if the non-deterministic algorithm is used, a `registrator` could reuse past signatures if they wanted to, as they are not rejected either on the execution layer or on the consensus layer.

Consider adding explicit checks against duplicate `depositDataRoots`, requiring the `registrator` to generate and use different signatures for each deposit.

Update: *Resolved by reverting if a duplicate deposit is detected. The deposit data root is no longer supplied by the user but calculated through the `merkleizePendingDeposit` function. While not currently a security risk, the `merkleizeSha256` function will revert for a tree with less than 2 leaves and cause unexpected behaviour if the number of leaves is not a power of 2.*

H-02 Validator Registration Frontrunning Can DoS the Strategy

The strategy utilizes the P2P API to create a private key and the corresponding key shares for validators to be registered on the SSV network. It is possible that the P2P API leaks the validator private key to a malicious actor, leading to the validator deposit being front-run with different withdrawal credentials and the protocol losing the initial deposit of 32 ETH, which is an accepted risk here.

To prevent further impact from deposits to such maliciously registered validator, the `verifyValidator` function in the `CompoundingValidatorManager` contract only allows the verification of a validator if the registered withdrawal credentials match the strategy's contract address. This ensures that the validator was honestly registered on the Beacon chain.

However, given the internal state management of the strategy contract, this additional check can permanently DoS the strategy in case of even one validator registration being front-run:

- Assuming a validator is being registered for the first time, the `registerSsvValidator` function is called. This function stores the validator state as `REGISTERED`.
- The `stakeETH` function is called, which adds the deposit to the pending deposit array (`deposits`). The validator's state is updated to `STAKED` state.
- Note that the `verifyDeposit` function for that deposit will not proceed until the validator is in a `VERIFIED` state as per this check.
- However, the `verifyValidator` function that updates the validator to `VERIFIED` state will also not proceed, because of this check, which requires the withdrawal credentials to match the strategy's address. Since the validator registration was front-run, the withdrawal credentials would be different. Thus, the validator will never reach a `VERIFIED` state, creating a deadlock.
- Consequently, there will always be an unverified deposit in the array which is already processed, making the `verifyBalance` function always revert.
- Since `verifyBalances` is an important function that updates the balances in the strategy, this scenario effectively bricks the whole strategy and requires a significant contract upgrade.

Consider implementing a way to update the states of the validators registered with different withdrawal credentials. One way is to remove their deposit from the `deposits` array to avoid the deadlock state. In addition, consider removing or marking any maliciously registered validator to prevent further contract interaction with it.

Update: *Resolved. The `verifyValidator` function will ensure a validator has been registered with the correct withdrawal credentials. If this was not the case, the validator will be marked as `INVALID` and the unverified deposit to it will be removed.*

Medium Severity

M-01 Deposits to Exiting Validators Can Cause Undercounting

On the Beacon chain, deposits to exiting validators are permitted, but they are postponed until the validator becomes withdrawable. Once the validator is withdrawable, balance updates on both the execution and consensus layers occur when the validator is reached through the automatic partial withdrawals sweep. This process can take varying amounts of time depending on the number and state of validators.

Similarly, postponed pending deposits are applied once the validator becomes withdrawable and is reached within the pending deposits list. Note that each pending deposit sweep occurs once per epoch, and processes at most `MAX_PENDING_DEPOSITS_PER_EPOCH` deposits, or until the deposit churn limit is reached. Hence, this process can also take varying amounts of time, depending on the state of the pending deposits list.

It is possible that deposits may be unknowingly made to an exiting validator, as the `stakeEth` function has no checks against such occurrences. Due to the full withdrawal behavior described above, a validator's balance may be updated to 0 before being increased again by a pending postponed deposit. A call to the verifyBalances function while the balance is 0 will mark the validator as `EXITED`, which is the expected behavior.

The postponed deposit may soon be processed but cannot be verified because the validator is in the `EXITED` state. This can cause DoS in the `verifyBalances` function. Moreover, if the DoS issue is resolved, undercounting may still occur: although the deposit is verified within the contract and should be reflected in the validator balances part of `verifyBalances`, it will not be accounted for because the validator has already exited and has been removed from `validatorsCount`. The deposit will only be included once it is swept through automatic partial withdrawals.

Consider handling the edge case where pending deposits are processed after the validator has exited and simultaneously deploying a mechanism to prevent deposits to validators that are in the exiting state.

Update: *The DoS vector was mitigated by only marking a validator as `EXITED` if the validator's consensus layer balance is 0 and there are no pending deposits to this validator. The undercounting is mostly no longer possible due to changes to the `verifyBalances` function, which now counts the total value in both pending deposits and validator balances through merkle proofs against the beacon state. However, a small, temporary undercounting is still possible due to depositing to an exiting validator. The Origin Protocol team has accepted this risk, as acknowledged here.*

M-02 Leaked or Malicious `registrator` Can Damage the Strategy

The `registrator` role can perform multiple privileged actions within the `CompoundingStakingSSVStrategy` contract, such as registering a new validator in the SSV network, depositing ETH from the strategy to validators or triggering a snap balance. Note that the `registrator` role is not as trusted as the governor. As such, the risks of misusing the `registrator` role should be limited as much as possible. Several such risks have been listed below along with ways to mitigate them:

- **Stealing Strategy Funds**: A compromised `registrator` could leverage the `registerSsvValidator` function to register validators not belonging to Origin and with different withdrawal credentials already set on the Beacon chain. Subsequently, the `registrator` could deposit 32 ETH to each of the validators, effectively stealing deposited WETH from the strategy. Consider adding a mechanism to limit the attack vector, such as only allowing a deposit of 1 ETH before the validator is verified to have the correct withdrawal credentials.

- **DoS Through Deposit Lifecycle Accounting**: Like the above case, depositing to a validator whose withdrawal credential does not match the strategy contract's address will cause the `verifyValidator` function to revert. Without the possibility of verifying the validator, the strategy can not verify the deposit, which will cause a permanent DoS of the `verifyBalances` function. Consider adding an access-controlled mechanism to remove any maliciously created deposits that can not be verified due to having different withdrawal credentials.

- **DoS Through Out-of-Gas**: The `verifyBalances` function spends gas proportionally to how many deposits are still pending and how many validators are verified and non-exited. This creates an attack vector where the `registrator` could cause permanent reverts of the `verifyBalances` function if the deposits and active validator arrays grow too big. Consider adding a mechanism to limit the amount of both pending deposits and active validators.

- **Stalling the `snapBalances` and `verifyBalances` Workflow**: The combination of `snapBalances` and `verifyBalances` is vital for the strategy, as it is the mechanism through which the vault determines the current value within the strategy. Stalling this mechanism can result in undercounting or overcounting the funds within the strategy. Consider adding a trusted entity that could trigger this workflow, such as the governor.

- **Locked ETH Within the Execution Layer Deposit Contract**: The consensus layer will validate a deposit's signature only for the first deposit to a validator. If this signature is invalid, the consensus layer simply returns, leaving all the deposited ETH within the execution-layer deposit contract. Consider adding a check that requires a deposit's BLS signature to be valid whenever the first deposit for a validator is made.

Consider iterating on the above suggestions in order to reduce the risks of a misbehaving `registrator`.

Update: *Partially resolved.*

> *The first bullet point was addressed by requiring the first deposit to a validator to be exactly 1 ETH, limiting the amount of funds that could be stolen. Additionally, a `firstDeposit` flag was added, which allows at most 1 deposit to an unverified validator, effectively limiting the total attack vector to 1 ETH.*
>
> *The second bullet point was addressed by marking a validator as `INVALID` if the withdrawal credentials do not match the ones enforced by the strategy (validator type 2, padding of 11 bytes of 0, address of the strategy as withdrawal address). Once `INVALID`, functionality can no longer be accessed for this validator, and all pending deposits to it are removed from accounting. When removing an invalid deposit, the `lastVerifiedEthBalance` variable is updated, in case the deposit was counted in the last `verifyBalances` call. Note that if `verifyBalances` had not accounted for the deposit, a slight undercount of 1 ETH can happen, which will be corrected in the next iteration of `verifyBalances`.*
>
> *The third bullet point was addressed by limiting the amount of pending deposits and of verified validators. Note that the `verifyBalances` function of the final version of the system does a secondary loop for each validator, and hence will consume substantially more gas. Hence, we advise that the maximum number of verified validators should be updated accordingly. Lastly, note that the `MAX_VERIFIED_VALIDATORS` check is off by 1, and will allow 1 less verified validators than expected.*
>
> *The fourth bullet point was addressed by removing the access control of the `snapBalances` function. Note that fully removing the access control of the `snapBalances` function enlarges the attack vector, if any method to manipulate the strategy's balance up and down is discovered - it would allow an attacker to manipulate the rebalancing mechanism of the vault multiple times within a short timespan.*
>
> *The fifth bullet point was not addressed.*

Low Severity

L-01 Variable Names Too Similar

Similar variable names make the code challenging to read and maintain, and can introduce confusion during the auditing process.

The `VALIDATOR_HEIGHT` name within `BeaconProofsLib.sol` is too similar to `VALIDATORS_HEIGHT`.

Consider renaming the `VALIDATORS_HEIGHT` constant to `VALIDATORS_LIST_HEIGHT`. Similarly, consider renaming the `VALIDATOR_HEIGHT` constant to `VALIDATOR_STRUCT_HEIGHT`. Doing so will help improve code clarity and maintainability.

Update: *Resolved in pull request #2625.*

L-02 `verifyBalances` Can Revert for a Pending Deposit

The `verifyBalances` function of the `CompoundingValidatorManager` contract verifies the balances of active validators. It performs a check that requires all unverified deposits to still be in the pending deposits list of the Beacon chain. However, when a deposit's `depositSlot` value is equal to `firstPendingSlot`, it is incorrectly considered verifiable in the smart contracts, even though it has not yet been processed on the Beacon chain. This creates a scenario where a pending deposit in the first position of `depositList` can cause the function to revert.

To address this edge case, consider updating the check to use `<=` instead of `<`.

Update: *Resolved. The issue was deemed invalid. The Origin Protocol team stated:*

> *If the slot of the first pending deposit is the same as the slot of the strategy's deposit, then the strategy does not for certain if the deposit was processed or not. It's safer to assume the worst and revert `verifyBalances`. If the strategy assumes the deposit has not been processed when it has been processed on the beacon chain, then `verifyBalances` will double count the deposit as the validator balance will have increased. If the strategy assumes the deposit has been processed but it has not yet*

> *been processed on the beacon chain, then `verifyBalances` will under count the deposit.*

L-03 The `ValidatorWithdraw` Event Can Emit Misleading Values

The `ValidatorWithdraw` event emits the amount submitted for withdrawal. However, this amount can mislead observers, as it does not reflect the amount that will actually be withdrawn on the consensus layer:

- If `amount` is 0, this signals a full exit, meaning the entire amount of the validator will be withdrawn (i.e. `is_full_exit_request = amount == FULL_EXIT_REQUEST_AMOUNT`).
- If the amount after withdrawal will be lower than the minimum activation balance (32 ETH), `amount` will be clamped such that at least 32 ETH are left (i.e., `to_withdraw = min(state.balances[index] - MIN_ACTIVATION_BALANCE - pending_balance_to_withdraw, amount)`).

Consider documenting the aforementioned edge case to avoid confusion for off-chain observers.

Update: *Resolved. The Origin Protocol team stated:*

> *The `validatorWithdrawal` Natspec states:*

```
/// @notice Request a full or partial withdrawal from a validator.
/// A zero amount will trigger a full withdrawal.
/// If the remaining balance is < 32 ETH then only the amount in excess of 32 ETH will
be withdrawn.
```

I think this explains what the withdrawal amount represents.

L-04 The `stakeEth` Function Does Not Leverage ETH That Exists in the Contract

The `stakeEth` function permits the `registrator` to deposit into the Beacon chain contract by specifying the amount of ETH to deposit. The contract requires a corresponding balance of WETH, which is converted to ETH and deposited. It is important to note that the contract might already have ETH within it due to automatic partial withdrawals, execution rewards, voluntary partial withdrawals, or exits due to slashings. This ETH is not taken into consideration, and in order to leverage it for new deposits, the vault will have to first withdraw it and then deposit it as WETH. While this process does not present a security risk, it unnecessarily involves additional steps, which increase the potential for errors.

Consider altering the `stakeEth` function such that it leverages the ETH balance of the contract and only converts any missing amount from WETH. Alternatively, consider adding an access-controlled function to convert existing ETH into WETH.

Update: *No code changes were made, the feature suggestion was acknowledged. The Origin Protocol team stated:*

> *The Vault can call `withdraw` or `withdrawAll` on the strategy to convert all the ETH to WETH. These are called by the Vault's `withdrawFromStrategy` or `withdrawAllFromStrategy` functions that are callable by the Strategist. The withdraw amount could be very small leaving most of the ETH as WETH in the contract. Or the Strategist can call `depositToStrategy` on the Vault to send the withdrawn WETH back to the strategy contract. I did consider having `stakeEth` always convert all ETH to WETH, but any significant amount of ETH will be from validator withdrawals so will get picked up with the strategy's `withdraw` function. Consensus rewards compound into the validator. Execution rewards go to the strategy and can be swept out with `withdraw` to the Vault.*

L-05 Lack of Input Validations

Throughout the codebase, multiple instances of missing input validations were identified:

- In the `CompoundingValidatorManager` contract:
    - The constructor does not verify any of the input parameters for emptiness.
    - The `registerSsvValidator` function does not validate the `publicKey` for emptiness.
    - The `stakeEth` function does not validate the `ValidatorStakeData` struct for emptiness.
    - The `removeSsvValidator` function does not validate the `publicKey` for emptiness.
- In the `CompoundingStakingSSVStrategy` contract, the constructor does not verify the `vaultAddress` to be non-empty and the `platformAddress` to be empty, since no platform is used for the `CompoundingStakingSSVStrategy` contract.

Consider adding proper input validations as required to avoid any unintended behavior in the contracts.

Update: *Acknowledged, not resolved. The Origin Protocol team stated:*

> *The contract size limit meant a lot of the emptiness checks were removed. The original emptiness checks came from early versions of web3.js which used a zero address as a default. That hasn't been like that for many years now so the value of checking emptiness has diminished.*

Notes & Additional Information

N-01 Valid Single-Leaf Proofs Are Always Rejected

The `processInclusionSHA256` function of the `Merkle` library contract rejects a Merkle proof whose length is zero even though an empty proof is perfectly valid for a Merkle tree that consists of a single leaf (i.e., the root equals the leaf itself). However, because of this check (`proof.length != 0`), users can never prove inclusion when the tree has only one element.

The `CompoundingStakingSSVStrategy` strategy utilizing this library is not vulnerable as the proof lengths will always be greater than zero for beacon-chain-related operations. Nonetheless, consider documenting this pitfall if the library is planned to be integrated with other strategies as well.

Update: *Acknowledged, not resolved. The Origin Protocol team stated:*

> *I think it is ok to reject zero length Merkle proofs in our implementation.*

N-02 Missing Named Parameters in Mappings

Since Solidity 0.8.18, mappings can include named parameters to provide more clarity about their purpose. Named parameters allow mappings to be declared in the form `mapping(KeyType KeyName? => ValueType ValueName?)`. This feature enhances code readability and maintainability.

Throughout the codebase, multiple instances of mappings without named parameters were identified:

- The `deposits`, `validatorState` and `snappedBalances` state variables in the `CompoundingValidatorManager` contract
- The `validatorsStates` state variable in the `ValidatorRegistrator` contract

Consider adding named parameters to mappings in order to improve the readability and maintainability of the codebase.

Update: *Acknowledged, not resolved. The Origin Protocol team stated:*

> *The version of the Solidity linter doesn't support mapping named parameters.*

N-03 Multiple Function Do Not Respect the Checks Effects Interactions (CEI) Pattern

Throughout the codebase, multiple instances of functions that interact with non-Origin contracts but do not follow the CEI pattern as per the Solidity security considerations were identified:

- The `stakeETH` function
- The `removeSsvValidator` function

While no security risks were identified as the external contracts are known and are not expected to change, this is not a guarantee in the future. As such, consider following the CEI pattern as much as possible to avoid unintended behavior.

Update: *Resolved in pull request #2634.*

N-04 Unused Code

Throughout the codebase, multiple instances of unused code were identified:

- The `SourceStrategyAdded` event
- The `BLOCK_NUMBER_GENERALIZED_INDEX` constant

To improve the overall clarity and maintainability of the codebase, consider removing any unused code.

Update: *Resolved in pull request #2626. The Origin Protocol team stated:*

> *Thanks. I've removed the `SourceStrategyAdded` event. `BLOCK_NUMBER_GENERALIZED_INDEX` was removed in the big refactoring PR.*

N-05 Lack of Security Contact

Providing a specific security contact (such as an email or ENS name) within a smart contract significantly simplifies the process for individuals to communicate if they identify a vulnerability in the code. This practice is quite beneficial as it permits the code owners to dictate the communication channel for vulnerability disclosure, eliminating the risk of miscommunication or failure to report due to a lack of knowledge on how to do so. In addition, if the contract incorporates third-party libraries and a bug surfaces in those, it becomes easier for their maintainers to contact the appropriate person about the problem and provide mitigation instructions.

Throughout the codebase, multiple instances of contracts missing a security contact were identified:

- The `BeaconProofs` contract
- The `CompoundingStakingSSVStrategy` contract
- The `FeeAccumulator` contract
- The `NativeStakingSSVStrategy` contract

Consider adding a NatSpec comment containing a security contact above each contract definition. Using the `@custom:security-contact` convention is recommended as it has been adopted by the OpenZeppelin Wizard and the ethereum-lists.

Update: *Acknowledged, not resolved. The Origin Protocol team stated:*

> *This is not something Origin Protocol has been doing.*

N-06 `_disableInitializers()` Not Being Called from Initializable Contract Constructors

In a proxy pattern, an implementation contract allows anyone to call its `initialize` function. While not a direct security concern, preventing the implementation contract from being initialized is important, as this could allow an attacker to take over the contract.

Throughout the codebase, multiple instances of initializable contracts where `_disableInitializers()` is not called in the constructor were identified:

- The initializable contract `CompoundingStakingSSVStrategy` within the `CompoundingStakingSSVStrategy.sol` file
- The initializable contract `NativeStakingSSVStrategy` within the `NativeStakingSSVStrategy.sol` file

Consider calling `_disableInitializers()` in initializable contract constructors to prevent malicious actors from front-running initializations.

***Update:** Resolved at commit 57951f2 in a different manner, consistent with the rest of the codebase. The Origin Protocol team stated:*

> *We have done with a different approach but achieves the same outcome of preventing anyone from calling `initialize` on the strategy's implementation contract. The `CompoundingStakingSSVStrategy` constructor now has:* `// Make sure nobody owns the implementation contract` `_setGovernor(address(0));`

This prevents anyone calling `initialize` on the implementation contract as it has a `onlyGovernor` modifier.

N-07 Code Simplifications

Throughout the codebase, multiple opportunities for code simplification were identified, some of which even yield gas savings:

- The `processInclusionProofSha256` function does the same division in both the `if` and `else` branches. Consider moving the division outside the branches, such that it applies regardless of the condition.
- The `_asset` parameter of the `_withdraw` functions of both the `NativeStakingSSVStrategy` and `CompoundingStakingSSVStrategy` contracts is redundant, as the only supported asset is WETH. Consider removing the parameter and hardcoding the withdrawal of WETH.
- In the `_snapBalances` function, consider converting the current `block.timestamp` to `uint64` only once and caching this value.
- This casting of `validatorBalanceGwei` to `uint256` is redundant, as the variable is already a `uint256`. Consider removing it.

- The `BalancesSnapped` event redundantly emits the `block.timestamp`, which can be inferred off-chain from the block in which the event resides. Consider simplifying the event by removing the emission of the timestamp.
- The `VALIDATOR_STATUS` enum does not follow the Solidity naming convention. Consider renaming the enum in the CapWords style.
- The `VALIDATOR_STATUS` and `DepositStatus` enums are defined below the data structures within which they are used, negatively affecting readability. Consider defining these enums above the data structures for improved readability.
- The `getPendingDeposits` function does not cache the length of the `depositsRoots` array. Consider caching the array length to save gas whenever this function is called during external function calls.

Consider applying the above suggestions to decrease code size, increase legibility and achieve small gas savings.

Update: *Resolved in pull request #2631.*

N-08 Missing, Incorrect, or Misleading Documentation

Throughout the codebase, multiple instances of missing, incorrect, or misleading documentation were identified:

- Line 423 of the `CompoundingValidatorManager` contract should state that `ssvAmount` is the amount to withdraw from the cluster.
- Line 289 of the `ValidatorRegistrator` contract should state that the validator is checked to be staked.
- Line 32 of the `CompoundingValidatorManager` contract should state `decreased to` instead of `increased to`.
- Line 378 of the `CompoundingValidatorManager` contract does not match the context of the `removeSsvValidator` function.
- The `lastSnapTimestamp` and `lastVerifiedEthBalance` state variables are missing documentation similar to other state variables.
- The `PartialWithdrawal` library is missing NatSpec comments for its functions.
- The TODO comment in line 93 of the `ValidatorRegistrator` contract is vague, and it is unclear to which part of the code it refers to. Consider adding more details to this TODO comment and tracking it inside the project's issue backlog.

Consider implementing the above suggestions to increase the clarity and maintainability of the codebase.

Update: *Resolved in pull request #2632, commit 303b6e9 and commit 2f588d2. The second and last bullet points were deemed out of scope and hence, not resolved.*

Recommendations

Design Considerations

Various design considerations were identified during the audit. While not necessarily a security risk, they should be taken into consideration as they limit functionality or could cause breaking changes:

- The consensus layer does not provide an abstract API to retrieve the positions of certain fields within the Beacon block root. Hence, these positions need to be manually computed and hardcoded. It is likely that with subsequent Ethereum upgrades, such as Fusaka, the structure and shape of the state tree will change, and hence, none of the indices will be correct anymore. This will cause all Merkle proofs to fail. As such, the Origin team should keep themselves up-to-date with all Ethereum developments and ensure that the code is updated and audited before the introduction of any major upgrade with breaking changes.
- The balance of the strategy is calculated based on the last successful run of the `verifyBalances` function, which is scheduled to be called once a day. Note that consensus layer balances can change due to many factors such as deposits, withdrawals, consolidations, penalties, slashings and rewards. Hence, to have a more precise and recent view of the balance, it is recommended to run the `verifyBalances` more often, such as several times a day.

Monitoring Recommendations

Given the nature of the codebase, the entities involved, and the tasks performed, enhanced monitoring is recommended:

- Consider monitoring the state of all validators, especially for unexpected slashing events that would create a loss for the strategy.
- Consider keeping track and ensuring that the `registrator` does not deposit to the validators that are `exiting`, as this will create a postponed deposit that will only be processed once the validator is exited.

Conclusion

The code under review introduces the `CompoundingStakingSSVStrategy` contract, which is an evolution of the `NativeStakingSSVStrategy` contract, better suited for the post-Pectra Beacon chain. The tracking of the strategy's balance is now more precise, counted, and validated through Merkle proofs against the Beacon chain root posted on the execution layer.

After thoroughly reviewing the pull request and the accompanying documentation, one critical- and several high- and medium-severity issues were identified. In addition, several low-severity issues were reported, and various recommendations to improve code quality were made.

The strategy's code was found to incorrectly assume the strictly increasing order of the Beacon chain pending deposits list. As a result, most functionality based on this assumption has been refactored. Moreover, multiple DoS attack vectors have surfaced as a result of the difficulty of syncing the states of the execution and consensus layers, together with the deposit and validator states of the smart contracts.

The fixes were comprehensive, including a significant redesign of the `verifyDeposit` and `verifyBalances` functions, along with remediation of findings from a third-party audit completed prior to our fix review. Although unlikely, several temporary denial-of-service and under-accounting possibilities still remain. The Origin Protocol team has acknowledged some of them as accepted risks. Any further changes will require careful review and deep implementation-level understanding of the consensus layer. Given the interconnectivity of states between the execution and consensus layers, the complexity of the smart contract accounting and the numerous state transitions, we recommend further code review to strengthen the codebase's security.

Throughout the audit, the Origin team was extremely responsive and collaborative, promptly addressing any questions posed and providing valuable insights. The provided technical documentation and team discussions were very helpful, which greatly assisted the audit team with understanding the `CompoundingStakingSSVStrategy` contract and all its integrations with the Beacon chain.