Origin Protocol

Compounding Staking Strategy

Security Assessment Report

Version: 2.1

September, 2025

Contents

Introduction

Sigma Prime was commercially engaged to perform a time-boxed security review of the Origin Protocol components in scope. The review focused solely on the security aspects of the Solidity implementation of the contracts, though general recommendations and informational comments are also provided.

Disclaimer

Sigma Prime makes all effort but holds no responsibility for the findings of this security review. Sigma Prime does not provide any guarantees relating to the function of the components in scope. Sigma Prime makes no judgements on, or provides any security review, regarding the underlying business model or the individuals involved in the project.

Document Structure

The first section provides an overview of the functionality of the Origin Protocol components contained within the scope of the security review. A summary followed by a detailed review of the discovered vulnerabilities is then given which assigns each vulnerability a severity rating (see Vulnerability Severity Classification), an *open/closed/resolved* status and a recommendation. Additionally, findings which do not have direct security implications (but are potentially of interest) are marked as *informational*.

The appendix provides additional documentation, including the severity matrix used to classify vulnerabilities within the Origin Protocol components in scope.

Overview

Origin Protocol is a suite of DeFi products designed to increase economic opportunity through innovative blockchain technologies, including liquid staking tokens and yield-generating stablecoins.

The compounding staking strategy represents a novel approach to Ethereum validator staking that uses compounding (`0x02` withdrawal credentials) validators to reduce operational costs and enables on-chain withdrawals without relying on third-party node operators. This strategy uses Merkle proofs to verify validator operations on the beacon chain and leverages Ethereum's Pectra upgrade features to create a more secure and efficient method of validator management.

Security Assessment Summary

Scope

The review was conducted on the files hosted on the OriginProtocol/origin-dollar repository.

The scope of this time-boxed review was strictly limited to the following files, assessed at commit f078483:

- `contracts/contracts/strategies/NativeStaking/CompoundingStakingSSVStrategy.sol`
- `contracts/contracts/strategies/NativeStaking/CompoundingStakingView.sol`
- `contracts/contracts/strategies/NativeStaking/CompoundingValidatorManager.sol`
- `contracts/contracts/beacon/BeaconProofs.sol`
- `contracts/contracts/interfaces/IBeaconProofs.sol`
- `contracts/contracts/beacon/BeaconProofsLib.sol`
- `contracts/contracts/beacon/BeaconRoots.sol`
- `contracts/contracts/beacon/PartialWithdrawal.sol`
- `contracts/contracts/beacon/Merkle.sol`
- `contracts/contracts/beacon/Endian.sol`

The fixes of the identified issues were assessed at commit 9518a13.

Note: third party libraries and dependencies were excluded from the scope of this assessment.

Approach

The security assessment covered components written in Solidity.

The manual review focused on identifying issues associated with the business logic implementation of the contracts. This includes their internal interactions, intended functionality and correct implementation with respect to the underlying functionality of the Ethereum Virtual Machine (for example, verifying correct storage/memory layout).

Additionally, the manual review process focused on identifying vulnerabilities related to known Solidity anti-patterns and attack vectors, such as re-entrancy, front-running, integer overflow/underflow and correct visibility specifiers.

For a more detailed, but non-exhaustive list of examined vectors, see [1, 2].

To support the Solidity components of the review, the testing team may use the following automated testing tools:

- Aderyn: `https://github.com/Cyfrin/aderyn`

- Slither: `https://github.com/trailofbits/slither`
- Mythril: `https://github.com/ConsenSys/mythril`

Output for these automated tools is available upon request.

Coverage Limitations

Due to the time-boxed nature of this review, all documented vulnerabilities reflect best effort within the allotted, limited engagement time. As such, Sigma Prime recommends to further investigate areas of the code, and any related functionality, where majority of critical and high risk vulnerabilities were identified.

Findings Summary

The testing team identified a total of 12 issues during this assessment. Categorised by their severity:

- Critical: 1 issue.
- High: 5 issues.
- Medium: 1 issue.
- Low: 3 issues.
- Informational: 2 issues.

Detailed Findings

This section provides a detailed description of the vulnerabilities identified within the Origin Protocol components in scope. Each vulnerability has a severity classification which is determined from the likelihood and impact of each issue by the matrix given in the Appendix: Vulnerability Severity Classification.

A number of additional properties of the components, including optimisations, are also described in this section and are labelled as "informational".

Each vulnerability is also assigned a **status**:

- ***Open:*** the issue has not been addressed by the project team.
- ***Resolved:*** the issue was acknowledged by the project team and updates to the affected components(s) have been made to mitigate the related risk.
- ***Closed:*** the issue was acknowledged by the project team but no further actions have been taken.

Summary of Findings

ID	Description	Severity	Status
ORGN-01	Array Element Skipping During Deposit Removal Causes Balance Undercount And DoS	Critical	Resolved
ORGN-02	Insufficient Withdrawal Credentials Validation Allows Permanent Fund Loss	High	Resolved
ORGN-03	Deposit Balance Undercount Due To Missing Timestamp Validation In `verifyDeposit()`	High	Resolved
ORGN-04	Validator State Prematurely Set To `EXITING` On Failed Withdrawal Requests	High	Resolved
ORGN-05	Validator State Check In `verifyDeposit()` Causes Balance Verification DoS	High	Resolved
ORGN-06	Deposit Double-Counting Via Exiting Validator Timing	High	Resolved
ORGN-07	Validator Deposit Front-Running Leads To Balance Overstatement	Medium	Resolved
ORGN-08	Exiting Validator Deposits Cause `verifyBalances()` DoS When Queue Is Empty	Low	Resolved
ORGN-09	Verification Epoch Calculation Off-By-One Error Causes Temporary DoS	Low	Resolved
ORGN-10	First Deposit Validator Exit Check Causes Deposit Verification DoS	Low	Resolved
ORGN-11	Missing Bounds Validation On Validator Index In `verifyValidator()`	Informational	Resolved
ORGN-12	Miscellaneous General Comments	Informational	Closed

ORGN-01	Array Element Skipping During Deposit Removal Causes Balance Undercount And DoS		
Asset	contracts/strategies/NativeStaking/CompoundingValidatorManager.sol		
Status	**Resolved:** See Resolution		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

The `verifyBalances()` function contains a critical array iteration flaw that simultaneously causes denial-of-service conditions and balance accounting errors, resulting in both system unavailability and financial miscalculations.

The `CompoundingValidatorManager` contract's `verifyBalances()` function suffers from a dual-impact vulnerability in its deposit processing loop. The function iterates through the `depositList` array whilst simultaneously removing elements during iteration, creating two distinct but related failure modes: array index out-of-bounds access leading to transaction reverts (DoS), and skipped array elements causing incorrect balance calculations (undercounting).

The vulnerability manifests when `_removeDeposit()` is called for validators in the `ValidatorState.EXITED` state. This function uses a swap-and-pop pattern that moves the last array element to the current index and decrements the array length. However, the loop implementation creates both DoS conditions and accounting errors through its flawed iteration pattern.

The vulnerable code pattern caches the array length before iteration and continues with a fixed loop counter:

contracts/strategies/NativeStaking/CompoundingValidatorManager.sol::verifyBalances()

```
uint256 depositsCount = depositList.length;  // @audit Cached before loop

// ... snip ...

if depositsCount > 0 {

    // ... snip ...

    for (uint256 i = 0; i < depositsCount; ++i) {
        uint256 depositID = depositList[i];
        DepositData memory depositData = deposits[depositID];

        // ... snip ...

        // Remove the deposit if the validator has exited.
        if (validator[depositData.pubKeyHash].state == ValidatorState.EXITED) {
            _removeDeposit(depositID, depositData);  // @audit Modifies array during iteration

            emit DepositValidatorExited(
                depositID,
                uint256(depositData.amountGwei) * 1 gwei
            );

            continue;  // @audit Skips the moved element at current index
        }

        // Convert the deposit amount from Gwei to Wei and add to the total
        totalDepositsWei += uint256(depositData.amountGwei) * 1 gwei;
    }
}
```

The `_removeDeposit()` function implements a swap-and-pop mechanism that creates the skipping condition:

contracts/strategies/NativeStaking/CompoundingValidatorManager.sol::_removeDeposit()

```
function _removeDeposit(uint256 depositID, DepositData memory deposit)
    internal
{
    // After verifying the proof, update the contract storage
    deposits[depositID].status = DepositStatus.VERIFIED;
    // Move the last deposit to the index of the verified deposit
    uint256 lastDeposit = depositList[depositList.length - 1];
    depositList[deposit.depositIndex] = lastDeposit;  // @audit Element moved to current position
    deposits[lastDeposit].depositIndex = deposit.depositIndex;
    // Delete the last deposit from the list
    depositList.pop();  // @audit Array length reduced
}
```

When this occurs, the moved element is never processed in subsequent loop iterations because the loop counter has already advanced past its new position. This results in skipped deposits not being included in the `totalDepositsWei` calculation, causing `lastVerifiedEthBalance` to be understated by the amount of each skipped deposit.

Additionally, the cached `depositsCount` value creates an index out-of-bounds access. Since `depositsCount` is set to the initial `depositList.length` before the loop begins, but each `_removeDeposit()` call reduces the actual array length via `depositList.pop()`, the loop may eventually attempt to access `depositList[i]` where `i >= depositList.length`. This would cause the transaction to revert with an index out-of-bounds error, resulting in a denial-of-service condition that prevents the `verifyBalances()` function from completing successfully.

This issue is classified as high impact because it causes both direct financial accounting errors and denial-of-service conditions. The financial impact stems from skipped deposits causing incorrect balance calculations, while the DoS impact occurs when a validator exit causes the loop to exceed the shortened array bounds. The `lastVerifiedEthBalance` becomes incorrect until the next successful `verifyBalances()` call, and in severe cases with at least one validator exit, the function may become completely non-functional.

The likelihood is assessed as high since this occurs during normal validator lifecycle management when multiple deposits exist and at least one validator with a pending deposit reaches the `ValidatorState.EXITED` state whilst its deposit is not the last element in the array. No malicious intervention is required for this vulnerability to manifest.

Recommendations

Modify the iteration pattern to handle array modifications correctly by using one of the following approaches:

1. Iterate backwards to avoid index out-of-bounds issues and skipping elements
2. Perform dynamic length checking and index adjustment whenever a deposit is removed

Resolution

The development team has implemented a significant redesign and refactor of the `CompoundingValidatorManager` contract according to the testing team's recommendations to address issues relating to tracking pending deposits on the beacon chain. More details can be found in the ORGN-06 finding resolution.

As a result, this issue is no longer applicable and has been resolved as of the retesting commit 9518a13 as `verifyBalances()`

no longer removes deposits from the `depositList` array.

ORGN-02	Insufficient Withdrawal Credentials Validation Allows Permanent Fund Loss		
Asset	`contracts/beacon/BeaconProofsLib.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

The `verifyValidator()` function accepts validators with invalid withdrawal credentials prefixes, enabling permanent fund loss through validators that can deposit ETH but never withdraw from the beacon chain.

According to the Electra Ethereum consensus specifications, valid withdrawal credentials must begin with `0x01` or `0x02`. Other prefixes render validators incapable of withdrawing funds from the beacon chain, despite successful deposits.

ethereum/consensus-specs/specs/electra/beacon-chain.md

```
def is_fully_withdrawable_validator(validator: Validator, balance: Gwei, epoch: Epoch) -> bool:
    """
    Check if ``validator`` is fully withdrawable.
    """
    return (
        # [Modified in Electra:EIP7251]
        has_execution_withdrawal_credential(validator)
        and validator.withdrawable_epoch <= epoch
        and balance > 0
    )

# ... snip ...

def is_partially_withdrawable_validator(validator: Validator, balance: Gwei) -> bool:
    """
    Check if ``validator`` is partially withdrawable.
    """
    max_effective_balance = get_max_effective_balance(validator)
    # [Modified in Electra:EIP7251]
    has_max_effective_balance = validator.effective_balance == max_effective_balance
    # [Modified in Electra:EIP7251]
    has_excess_balance = balance > max_effective_balance
    return (
        # [Modified in Electra:EIP7251]
        has_execution_withdrawal_credential(validator)
        and has_max_effective_balance
        and has_excess_balance
    )

# ... snip ...

def process_withdrawal_request(state: BeaconState, withdrawal_request: WithdrawalRequest) -> None:
    # ... snip ...

    # Verify withdrawal credentials
    has_correct_credential = has_execution_withdrawal_credential(validator)
    is_correct_source_address = (
        validator.withdrawal_credentials[12:] == withdrawal_request.source_address
    )
    if not (has_correct_credential and is_correct_source_address):
        return

    # ... snip ...
```

The `verifyValidator()` function in `BeaconProofsLib.sol` validates withdrawal credentials by extracting only the withdrawal address (last 20 bytes) without checking the required prefix.

contracts/beacon/BeaconProofsLib.sol::verifyValidator()

```
// Get the withdrawal address from the first witness in the pubkey merkle proof.
address withdrawalAddressFromProof;
// solhint-disable-next-line no-inline-assembly
assembly {
    // The first 32 bytes of the proof is the withdrawal credential so load it into memory.
    calldatacopy(0, proof.offset, 32)
    // Cast the 32 bytes in memory to an address which is the last 20 bytes.
    withdrawalAddressFromProof := mload(0) // @audit Only extracts address, ignores prefix
}
require(
    withdrawalAddressFromProof == withdrawalAddress, // @audit Missing prefix validation
    "Invalid withdrawal address"
);
```

A malicious registrator can front-run legitimate beacon chain deposits by submitting deposits with identical withdrawal addresses but invalid prefixes (e.g., `0x03`). The contract accepts these as valid since it only verifies the address portion. The beacon chain deposit contract accepts deposits with any prefix, but withdrawal validation occurs only during withdrawal attempts. This creates validators that appear operational but have permanently locked funds.

This issue is classified as high impact because it has the potential to permanently lose a significant amount of ETH. ETH deposited to validators with invalid prefixes become permanently unrecoverable. Since this vulnerability allows the validator to be verified, more than 32 ETH can be permanently lost. Furthermore, the invalid validators count towards the 48 validator limit (`MAX_VERIFIED_VALIDATORS`), preventing legitimate validator registrations, and cannot be removed by calling `removeSsvValidator()` , which can only remove `REGISTERED` , `EXITED` , or `INVALID` validators.

The likelihood is assessed as medium since it requires a malicious registrator with front-running capabilities. Since the registrator role is only semi-trusted, this is a realistic attack vector.

Recommendations

Add prefix validation to the withdrawal credentials verification logic. Extract the full 32-byte withdrawal credentials and validate the prefix before checking the address portion.

Resolution

The development team has implemented a check for the entire withdrawal credentials including the prefix.

The logic has also been simplified, avoiding the use of inline assembly.

This issue has been resolved in PRs #2644, #2645, and #2659.

ORGN-03	Deposit Balance Undercount Due To Missing Timestamp Validation In `verifyDeposit()`		
Asset	`contracts/strategies/NativeStaking/CompoundingValidatorManager.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: Medium	Likelihood: High

Description

The `verifyDeposit()` function allows systematic ETH balance undercounting due to missing timestamp validation, enabling removal of deposits that were processed after the current balance snapshot.

The `CompoundingValidatorManager` contract uses a balance verification system where `snapBalances()` records the contract's ETH balance and timestamp, and `verifyBalances()` calculates the total balance at that timestamp by summing the contract balance with validator balances and pending deposits. However, the `verifyDeposit()` function lacks crucial timestamp validation that prevents deposits processed after `lastSnapTimestamp` from being verified and removed from the pending deposits list.

This creates a systematic accounting error where:

1. `snapBalances()` records ETH balance at time `T` and sets `lastSnapTimestamp = T`
2. Deposits are processed on the beacon chain after time `T` but before `verifyBalances()` is called
3. `verifyDeposit()` allows these deposits to be verified and removed from `totalDepositsWei` calculation
4. The removed deposits were not included in validator balances at snapshot time `T`, causing an undercount

The current `verifyDeposit()` function only validates that the deposit was processed before the validator creation slot, but fails to check if the deposit was processed after the balance snapshot:

contracts/strategies/NativeStaking/CompoundingValidatorManager.sol::verifyDeposit()

```solidity
function verifyDeposit(
    uint256 depositID,
    uint64 depositProcessedSlot,
    // ... other parameters
) external {
    // ... validation logic
    require(deposit.slot < depositProcessedSlot, "Slot not after deposit");
    // @audit Missing: timestamp validation against lastSnapTimestamp
    require(
        depositProcessedSlot <= firstDepositValidatorCreatedSlot,
        "Invalid verification slots"
    );
}
```

This issue is classified as medium impact because although funds are not at risk, it affects the strategy's total balance reporting and could impact accounting accuracy across the system.

The likelihood is assessed as high since this occurs in normal operational flow when deposits are processed on the beacon chain between `snapBalances()` and `verifyBalances()` calls. Given the typical ~6.4 minute epoch duration and that `verifyDeposit()` is externally callable, this timing window provides ample opportunity for these deposits to be processed and verified.

Recommendations

Implement timestamp validation in the `verifyDeposit()` function to prevent deposits processed after the last balance snapshot from being verified.

contracts/strategies/NativeStaking/CompoundingValidatorManager.sol::verifyDeposit()

```
require(
    (_calcNextBlockTimestamp(depositProcessedSlot) <= lastSnapTimestamp) || lastSnapTimestamp == 0,
    "Deposit processed after last balance snapshot"
);
```

Resolution

The development team has added timestamp validation in the `verifyDeposit()` function as recommended.

This issue has been resolved in PR #2651.

ORGN-04	Validator State Prematurely Set To `EXITING` On Failed Withdrawal Requests		
Asset	`contracts/strategies/NativeStaking/CompoundingValidatorManager.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

The `validatorWithdrawal()` function prematurely sets the validator state to `EXITING` before confirming that the withdrawal request has been accepted by the beacon chain, creating a permanent DoS that prevents future staking and withdrawal operations.

The vulnerability occurs in the `validatorWithdrawal()` function in `CompoundingValidatorManager.sol`. When a full withdrawal is requested (indicated by `amountGwei == 0`), the contract immediately sets the validator state to `ValidatorState.EXITING` without waiting for confirmation that the beacon chain has accepted the withdrawal request:

contracts/strategies/NativeStaking/CompoundingValidatorManager.sol::validatorWithdrawal()

```
if (amountGwei == 0) {
    // Store the validator state as exiting so no more deposits can be made to it.
    validator[pubKeyHash].state = ValidatorState.EXITING; // @audit Premature state change
}
```

However, the beacon chain can reject withdrawal requests under several conditions:

1. If the validator is not active (`epoch < activation_epoch || epoch >= exit_epoch`)
2. If the validator is already exiting (`exit_epoch != FAR_FUTURE_EPOCH`)
3. If the validator has not been active long enough (`current_epoch < activation_epoch + SHARD_COMMITTEE_PERIOD` , which is approximately 27 hours after activation)
4. If the validator has pending partial withdrawal requests in the queue

When a withdrawal request is rejected by the beacon chain, the validator remains active and functional on the consensus layer, but the strategy contract believes it is exiting. This creates a critical mismatch between the actual validator state and the contract's perceived state.

The premature state change has severe consequences across multiple functions:

- `stakeEth()` requires validators to be in `REGISTERED` or `VERIFIED` state:

  contracts/strategies/NativeStaking/CompoundingValidatorManager.sol::stakeEth()

  ```
  require(
      (currentState == ValidatorState.REGISTERED ||
          currentState == ValidatorState.VERIFIED),
      "Not registered or verified"
  );
  ```

- `validatorWithdrawal()` requires validators to be in `VERIFIED` state:

  contracts/strategies/NativeStaking/CompoundingValidatorManager.sol::validatorWithdrawal()

  ```
  require(
      currentState == ValidatorState.VERIFIED,
      "Validator not verified"
  );
  ```

- `verifyDeposit()` also requires validators to be in `VERIFIED` state:

  contracts/strategies/NativeStaking/CompoundingValidatorManager.sol::verifyDeposit()

  ```
  require(
      strategyValidator.state == ValidatorState.VERIFIED,
      "Validator not verified"
  );
  ```

This issue is classified as high impact because the vulnerability can cause a permanent denial-of-service with no recovery mechanism. Once a validator is erroneously marked as `EXITING`, it cannot receive additional stakes, submit future withdrawal requests, or verify pending deposits. If the pending deposit has been processed on the beacon chain, `verifyBalances()` will revert as the deposit's slot number will be lower than the first pending deposit's slot number, but the deposit does not have a withdrawable epoch set.

The likelihood is assessed as medium because while the vulnerability requires specific conditions to trigger, although these conditions are realistic and can occur during normal operations. `validatorWithdrawal()` is only callable by the registrator, who could unknowingly trigger the vulnerability or do so with malicious intent.

Recommendations

Only set the validator state to `EXITING` after receiving confirmation that the beacon chain has accepted the withdrawal request.

Resolution

The development team has implemented the following changes in the following PRs:

1. PRs #2641 and #2667: Added the ability to request full withdrawals for `EXITING` validators
2. PRs #2662 and #2668: Added a new `ValidatorState.ACTIVE` state to prevent requesting full withdrawals for inactive validators

This issue has been resolved.

ORGN-05	Validator State Check In `verifyDeposit()` Causes Balance Verification DoS		
Asset	`contracts/strategies/NativeStaking/CompoundingValidatorManager.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

Exiting validators cannot verify deposits due to an overly restrictive state check in `verifyDeposit()`, preventing the setting of critical `deposit.withdrawableEpoch` values needed for balance verification and causing denial-of-service in `verifyBalances()`.

The `verifyDeposit()` function contains a state requirement that only allows validators in `VERIFIED` state to proceed with deposit verification:

contracts/strategies/NativeStaking/CompoundingValidatorManager.sol::verifyDeposit()

```
function verifyDeposit(
    bytes32 depositID,
    uint64 depositProcessedSlot,
    uint64 firstDepositValidatorCreatedSlot,
    FirstPendingDepositProofData calldata firstPendingDeposit,
    DepositValidatorProofData calldata strategyValidatorData
) external {
    // Load into memory the previously saved deposit data
    DepositData memory deposit = deposits[depositID];
    ValidatorData memory strategyValidator = validator[deposit.pubKeyHash];
    require(deposit.status == DepositStatus.PENDING, "Deposit not pending");
    require(
        strategyValidator.state == ValidatorState.VERIFIED, // @audit Overly restrictive check
        "Validator not verified"
    );
    // ... snip ...
}
```

This restriction prevents exiting validators from executing the critical code path that sets `deposit.withdrawableEpoch` when the validator is exiting:

contracts/strategies/NativeStaking/CompoundingValidatorManager.sol::verifyDeposit()

```
// If the validator is exiting because it has been slashed
if (strategyValidatorData.withdrawableEpoch != FAR_FUTURE_EPOCH) {
    // Store the exit epoch in the deposit data
    deposit.withdrawableEpoch = strategyValidatorData.withdrawableEpoch; // @audit Cannot be set for EXITING validators
    emit DepositToValidatorExiting(
        depositID,
        uint256(deposit.amountGwei) * 1 gwei,
        strategyValidatorData.withdrawableEpoch
    );
    validator[deposit.pubKeyHash].state = ValidatorState.EXITING;
    // Leave the deposit status as PENDING
    return;
}
```

The `deposit.withdrawableEpoch` field is essential for the balance verification logic in `verifyBalances()`, which uses it to determine whether exiting validator pending deposits are still in the queue and should still be counted in the strategy's total ETH after their slot number has passed in the deposit queue:

contracts/strategies/NativeStaking/CompoundingValidatorManager.sol::verifyBalances()

```
require(
    firstPendingDeposit.slot < depositData.slot ||
        (verificationEpoch < depositData.withdrawableEpoch &&
            depositData.withdrawableEpoch != FAR_FUTURE_EPOCH) || // @audit Fails when withdrawableEpoch is FAR_FUTURE_EPOCH
        validator[depositData.pubKeyHash].state == ValidatorState.EXITED,
    "Deposit likely processed"
);
```

When `deposit.withdrawableEpoch` remains at its default value of `FAR_FUTURE_EPOCH` (because `verifyDeposit()` cannot be called), the second condition fails, and unless the validator has fully exited, the entire `verifyBalances()` call reverts.

According to the Ethereum consensus specifications, deposits to exiting validators are postponed:

ethereum/consensus-specs/specs/electra/beacon-chain.md

```
elif is_validator_exited:
# Validator is exiting, postpone the deposit until after withdrawable epoch
deposits_to_postpone.append(deposit)
```

This confirms that deposits to exiting validators require special handling and should be verifiable to track their postponement status.

This issue is classified as high impact because it causes complete denial-of-service in critical balance verification functionality. The DoS persists until the validator fully exits, which can be an extended duration (the exit queue wait is ~15 days at time of writing). During this period, the strategy cannot update its ETH balance, leading to operational disruption and potential financial inconsistencies.

The likelihood is assessed as medium since the vulnerability occurs when a validator with pending deposits enters the exiting state, which is expected during normal staking operations. Validators can exit voluntarily or be slashed, making this scenario realistic and likely to occur during the protocol's lifecycle.

Recommendations

Modify the state check in `verifyDeposit()` to allow both `VERIFIED` and `EXITING` validators to proceed with deposit verification:

contracts/strategies/NativeStaking/CompoundingValidatorManager.sol::verifyDeposit()

```
require(
    strategyValidator.state == ValidatorState.VERIFIED ||
    strategyValidator.state == ValidatorState.EXITING,
    "Validator not verified or exiting"
);
```

This change enables exiting validators to set the critical `deposit.withdrawableEpoch` field, allowing `verifyBalances()` to properly handle deposits to exiting validators according to the Ethereum consensus specification. The existing logic already handles the `EXITING` state appropriately by setting the withdrawable epoch and leaving the deposit status as pending.

Alternatively, replace the per-deposit `withdrawableEpoch` field with a per-validator `withdrawableEpoch` field. A separate function could be used to update the validator's `withdrawableEpoch` field.

Resolution

The development team has implemented the following changes:

- Modified the `verifyDeposit()` function to allow both `VERIFIED` and `EXITING` validators to proceed with deposit verification, and modified the `deposit.withdrawableEpoch` assignment to commit the change to contract state
- Modified `validatorWithdrawal()` to prevent validators with pending deposits from sending full withdrawal requests

This issue has been resolved in PR #2647 and #2652.

ORGN-06	Deposit Double-Counting Via Exiting Validator Timing		
Asset	`contracts/strategies/NativeStaking/CompoundingValidatorManager.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

A timing-based vulnerability allows deposit amounts to be double-counted in the strategy's verified ETH balance when validators initiate an exit after deposits have been processed but before their withdrawable epoch is reached.

The vulnerability arises from the interaction between `snapBalances()`, `verifyDeposit()`, and `verifyBalances()` functions when a validator initiates an exit after their deposit has been processed. When a deposit is processed on the beacon chain and included in a validator's balance, but the validator then enters an exiting state before verifying the deposit via `verifyDeposit()`, the processed deposit is not removed from `depositList`.

contracts/strategies/NativeStaking/CompoundingValidatorManager.sol::verifyDeposit()

```
if (strategyValidatorData.withdrawableEpoch != FAR_FUTURE_EPOCH) {
    // Store the exit epoch in the deposit data
    deposit.withdrawableEpoch = strategyValidatorData.withdrawableEpoch;

    emit DepositToValidatorExiting(
        depositID,
        uint256(deposit.amountGwei) * 1 gwei,
        strategyValidatorData.withdrawableEpoch
    );

    validator[deposit.pubKeyHash].state = ValidatorState.EXITING;

    // Leave the deposit status as PENDING
    // @audit The deposit is not removed even if it has been processed on the beacon chain
    return;
}
```

If balances are verified after the deposit has been processed but before the validator has reached its withdrawable epoch, the deposit is counted twice: once in the validator's balance and again as a pending deposit.

contracts/strategies/NativeStaking/CompoundingValidatorManager.sol::verifyBalances()

```
// @audit This check passes as the validator's withdrawable epoch has not passed at verificationEpoch
require(
    firstPendingDeposit.slot < depositData.slot ||
        (verificationEpoch < depositData.withdrawableEpoch &&
            depositData.withdrawableEpoch != FAR_FUTURE_EPOCH) ||
        validator[depositData.pubKeyHash].state == ValidatorState.EXITED,
    "Deposit likely processed"
);
```

The attack scenario unfolds as follows:

1. A pending deposit is processed on the beacon chain and added to the validator's balance.
2. The validator initiates an exit through a voluntary exit (BLS-signed message) or slashing, such that their `withdrawableEpoch != FAR_FUTURE_EPOCH`.

3. `snapBalances()` captures the validator balance including the processed deposit.
4. `verifyDeposit()` is called, setting the deposit's `withdrawableEpoch` but leaving the status as `PENDING` due to the early `return` statement.
5. `verifyBalances()` is called and the second condition `(verificationEpoch < depositData.withdrawableEpoch && depositData.withdrawableEpoch != FAR_FUTURE_EPOCH)` evaluates to true, allowing the deposit to pass validation.

The final balance calculation combines all components:

contracts/strategies/NativeStaking/CompoundingValidatorManager.sol::verifyBalances()

```
lastVerifiedEthBalance = SafeCast.toUint128(
    // @audit Both totalDepositsWei and totalValidatorBalance count the processed deposit
    totalDepositsWei + totalValidatorBalance + balancesMem.ethBalance
);
```

This vulnerability is classified as high impact because it results in direct balance overstatement that can affect yield calculations, share prices, and protocol accounting. The double-counted balance persists until the validator completes a full withdrawal after their withdrawable epoch has passed. Keep in mind that there is a period after the withdrawable epoch has passed but before the validator has withdrawn where `verifyBalances()` will revert as `verificationEpoch >= depositData.withdrawableEpoch`.

The likelihood is assessed as medium as it requires the validator to initiate an exit without calling `validatorWithdrawal()` in the strategy contract, as this would mark the validator as `EXITING` and the deposit would not be able to be verified. The validator would need to broadcast a BLS-signed voluntary exit message or be slashed.

Recommendations

Deposits that are not removed from `depositList` must require an inclusion proof to verify that they are still pending on the beacon chain.

Modify `verifyDeposit()` to verify the pending deposit inclusion proof if the validator is exiting before setting `deposit.withdrawableEpoch`. If no valid inclusion proof is provided, the deposit must be removed from `depositList`.

Resolution

During the retesting phase, the testing team identified that the proposed fix in PR #2647 did not fully address the vulnerability. Specifically, validator exits due to slashing or voluntary exits could still allow the attack scenario described above to play out.

Following extensive discussions between the development and testing teams, a comprehensive solution was developed that required a significant redesign and refactor of the existing `CompoundingValidatorManager` contract. The solution involved requiring inclusion proofs of all pending deposits in the `depositList` array to ensure they remain in the deposit queue at the snapshot in `verifyBalances()`, and removing the `_removeDeposit()` logic for exited validators from `verifyBalances()`. Instead, `verifyDeposit()` would handle deposit removal for exited validators by allowing deposit removal when the first deposit's slot number was greater than or equal to the validator's withdrawable epoch.

The testing team has also informed the development team that the refactored code will still cause a denial of service to `verifyBalances()` and `verifyDeposit()` if the attack scenario described in this finding occurs. There is no way to fix this issue without tracking the exact slots when the deposit has been processed, and when the validator initiates an exit, changes that would add significant complexity to the codebase as there is no easy method to detect both of these events. The DoS will last until the validator's withdrawable epoch passes, in which the processed deposit can finally be verified.

The development team has acknowledged this limitation with the following comment:

> *"This exploit is only possible if the validator exits voluntarily or is slashed on the beacon chain, since we already prevent EL-triggered exits in `validatorWithdrawal()` if the validator has a pending deposit. Hence, it requires the validator private key to be compromised or the validator to be slashed. The risk of compromised validator keys will be reduced as Origin no longer needs to store the encrypted validator keys with EL-triggered exits. The node operator is intended to discard the private key after it's been split in the SSV network, so if they can still initiate a voluntary exit, it is obvious they are malicious and so we know who to blame."*

The development team has resolved the issue by implementing this comprehensive solution in PRs #2658, #2669, and #2670.

ORGN-07	Validator Deposit Front-Running Leads To Balance Overstatement		
Asset	`contracts/strategies/NativeStaking/CompoundingValidatorManager.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

A malicious registrator can exploit validator deposit front-running to cause ETH balance overstatement by up to 32 ETH per attack, with the phantom balance persisting until the next verification cycle.

The contract implements a `firstDeposit` flag as a defence mechanism against validator deposit front-running attacks, where a malicious registrator could register validators with withdrawal credentials pointing to their own address instead of the strategy contract. When such an attack occurs, the 32 ETH deposit is effectively owned by the attacker, not by the strategy contract. However, the `verifyBalances()` function does not validate the `firstDeposit` flag before proceeding with balance verification, creating a timing window where stolen deposits are incorrectly counted towards the strategy's balance.

contracts/strategies/NativeStaking/CompoundingValidatorManager.sol::verifyBalances()

```
function verifyBalances(
    bytes32 snapBlockRoot,
    uint64 validatorVerificationBlockTimestamp,
    FirstPendingDepositProofData calldata firstPendingDeposit,
    BalanceProofs calldata balanceProofs
) external {
    // @audit Missing validation of firstDeposit flag allows balance verification with pending unverified deposits
    // Load previously snapped balances for the given block root
    Balances memory balancesMem = snappedBalances[snapBlockRoot];

    // ... snip ...

    for (uint256 i = 0; i < depositsCount; i++) {
        DepositData memory depositData = depositList[i];

        // ... snip ...

        // @audit Stolen deposit amounts are included in balance calculation
        // Convert the deposit amount from Gwei to Wei and add to the total
        totalDepositsWei += uint256(depositData.amountGwei) * 1 gwei;
    }

    // @audit Balance becomes overstated by including stolen deposits
    // Store the verified balance in storage
    lastVerifiedEthBalance = SafeCast.toUint128(
        totalDepositsWei + totalValidatorBalance + balancesMem.ethBalance
    );
}
```

This issue is classified as medium impact because the vulnerability results in direct balance overstatement of 32 ETH. This phantom balance could lead to incorrect accounting in dependent systems, potentially resulting in insolvency if users try to withdraw more funds than the strategy actually controls. The overstatement persists until the next balance verification cycle.

The likelihood is assessed as medium since the attack requires a malicious or compromised registrator, which is a semi-trusted role.

Recommendations

Implement validation of the `firstDeposit` flag at the beginning of the `verifyBalances()` function to prevent balance verification when there are pending unverified deposits that may be the result of front-running attacks.

Since new validator deposits are a frequent occurrence in the contract and may cause significant disruptions to balance verification, an alternative approach is to retroactively reduce `lastVerifiedEthBalance` when the malicious deposit is removed. Take extra precaution to ensure that the malicious deposit was counted in `lastVerifiedEthBalance` before `lastVerifiedEthBalance` is reduced. A potential solution involves tracking the slot number of the latest `verifyBalances()` call and comparing it to the slot number of the malicious deposit.

Resolution

The development team has implemented logic to adjust the `lastVerifiedEthBalance` value when a malicious deposit is removed.

This issue has been resolved in PR #2643.

ORGN-08	Exiting Validator Deposits Cause `verifyBalances()` DoS When Queue Is Empty		
Asset	`contracts/strategies/NativeStaking/CompoundingValidatorManager.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

The `verifyDeposit()` function fails to remove deposits for exiting validators, creating a scenario where the contract maintains deposit records whilst the beacon chain deposit queue is empty, causing a temporary denial-of-service in `verifyBalances()`.

When a validator is exiting (indicated by `withdrawableEpoch != FAR_FUTURE_EPOCH`), the `verifyDeposit()` function sets the deposit's `withdrawableEpoch` field and validator state to `EXITING`, but crucially leaves the deposit status as `PENDING` and returns early without removing the deposit from the contract's `depositList`:

contracts/strategies/NativeStaking/CompoundingValidatorManager.sol::verifyDeposit()

```
// If the validator is exiting because it has been slashed
if (strategyValidatorData.withdrawableEpoch != FAR_FUTURE_EPOCH) {
    // Store the exit epoch in the deposit data
    deposit.withdrawableEpoch = strategyValidatorData.withdrawableEpoch;

    emit DepositToValidatorExiting(
        depositID,
        uint256(deposit.amountGwei) * 1 gwei,
        strategyValidatorData.withdrawableEpoch
    );

    validator[deposit.pubKeyHash].state = ValidatorState.EXITING;

    // Leave the deposit status as PENDING
    return; // @audit Deposit not removed
}
```

This creates a logic inconsistency that manifests in `verifyBalances()`. The function expects that if the contract has deposits (`depositsCount > 0`), then the beacon chain deposit queue cannot be empty. However, when deposits belong to exiting validators, the queue becomes empty whilst the contract still tracks these deposits:

contracts/strategies/NativeStaking/CompoundingValidatorManager.sol::verifyBalances()

```
if (depositsCount > 0) {
    // Verify the slot of the first pending deposit matches the beacon chain
    bool isDepositQueueEmpty = IBeaconProofs(BEACON_PROOFS)
        .verifyFirstPendingDeposit(
            snapBlockRoot,
            firstPendingDeposit.slot,
            firstPendingDeposit.pubKeyHash,
            firstPendingDeposit.pendingDepositPubKeyProof
        );

    // If there are no deposits in the beacon chain queue then our deposits must have been processed.
    // If the deposits have been processed, each deposit will need to be verified with `verifyDeposit`
    require(!isDepositQueueEmpty, "Deposits have been processed"); // @audit DoS condition
}
```

An example DoS scenario unfolds as follows:

1. A deposit is made to a validator that subsequently exits (voluntary exit, or slashing)
2. `verifyDeposit()` is called for this exiting validator, which sets the deposit's `withdrawableEpoch` but leaves it as `PENDING`
3. The beacon chain deposit queue becomes empty as the validator's exit is processed
4. `verifyBalances()` is called with `depositsCount > 0` (due to the unremoved deposit) and `isDepositQueueEmpty = true`

This issue is classified as medium impact because it causes a denial of service of the critical `verifyBalances()` function, preventing balance accounting updates until either the deposit queue is no longer empty, or the validator fully withdraws.

The likelihood is assessed as low since the vulnerability occurs when deposits are made to validators that subsequently exit before the deposits are fully processed and `snapBalances()` is called when the deposit queue is empty. The deposit queue is unlikely to be empty.

Recommendations

Modify the `verifyDeposit()` function to properly remove deposits when the validator is exiting, rather than leaving them in `PENDING` state. A valid inclusion proof should be verified for the deposit if it is not removed from `depositList`.

Resolution

The development team has implemented a significant redesign and refactor of the `CompoundingValidatorManager` contract according to the testing team's recommendations to address issues relating to tracking pending deposits on the beacon chain. More details can be found in the ORGN-06 finding resolution.

As a result, this issue is no longer applicable and has been resolved as of the retesting commit 9518a13 as `verifyDeposit()` no longer keeps deposits for exiting validators and `verifyBalances()` no longer checks for an empty deposit queue.

ORGN-09	Verification Epoch Calculation Off-By-One Error Causes Temporary DoS		
Asset	`contracts/strategies/NativeStaking/CompoundingValidatorManager.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

The `verifyBalances()` function contains a timing calculation bug that causes temporary denial of service when called at epoch boundaries, preventing balance verification until re-called at a different time.

The verification epoch calculation is ahead by 1 of the actual verification epoch when `balancesMem.timestamp` is at the start of an epoch. This occurs because verification is performed against the **parent** beacon block root (previous slot), but the epoch calculation uses the current timestamp, creating a logical mismatch.

The root cause lies in the `verifyBalances()` function:

contracts/strategies/NativeStaking/CompoundingValidatorManager.sol::verifyBalances()

```
uint64 verificationEpoch = (SafeCast.toUint64(
    balancesMem.timestamp
) - BEACON_GENESIS_TIMESTAMP) / (SLOT_DURATION * SLOTS_PER_EPOCH);
```

This calculation determines the verification epoch using the current timestamp. However, the beacon proofs are verified against the parent beacon block root (previous slot), creating a temporal inconsistency.

The bug manifests in the require statement below:

contracts/strategies/NativeStaking/CompoundingValidatorManager.sol::verifyBalances()

```
require(
    firstPendingDeposit.slot < depositData.slot ||
        (verificationEpoch < depositData.withdrawableEpoch && // @audit Off-by-one here
            depositData.withdrawableEpoch !=
            FAR_FUTURE_EPOCH) ||
        validator[depositData.pubKeyHash].state ==
        ValidatorState.EXITED,
    "Deposit likely processed"
);
```

An example DoS scenario unfolds as follows:

1. A validator's `withdrawableEpoch` equals the current epoch (e.g., epoch 2)
2. `snapBalances()` is called at the start of epoch 2 (slot 64)
3. Verification uses the parent block root from slot 63 (epoch 1)
4. `verificationEpoch` is incorrectly calculated as epoch 2
5. The condition `verificationEpoch < withdrawableEpoch` (2 < 2) evaluates to false
6. `verifyBalances()` reverts with "Deposit likely processed"

This issue is classified as medium impact because it causes temporary service disruption of the balance verification functionality, which is critical for the system's operation. While no funds are at risk and the issue is self-resolving, it can prevent legitimate operations at specific timing windows.

The likelihood is assessed as low given that the vulnerability requires precise timing where `snapBalances()` is called at exact epoch boundaries when a validator becomes withdrawable.

Recommendations

Calculate the verification epoch using the parent slot's timestamp instead of the current timestamp to align with the actual beacon proof verification.

Keep in mind that if the parent slot was missed, then `snapBlockRoot` will belong to the last non-missed slot. It's not viable to efficiently detect when a slot has been missed in a smart contract, so this approach can still result in an inaccurate `verificationEpoch` if the first slot in an epoch was missed.

Resolution

The development team modified `verifyBalances()` to calculate the verification epoch using the parent slot's timestamp instead of the current timestamp in PR #2656.

Furthermore, the development team has implemented a significant redesign and refactor of the `CompoundingValidatorManager` contract according to the testing team's recommendations to address issues relating to tracking pending deposits on the beacon chain. More details can be found in the ORGN-06 finding resolution.

As a result, this issue is also no longer applicable and has been resolved as of the retesting commit 9518a13 as `verifyBalances()` no longer calculates the verification epoch.

ORGN-10	First Deposit Validator Exit Check Causes Deposit Verification DoS		
Asset	`contracts/strategies/NativeStaking/CompoundingValidatorManager.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Medium

Description

The `verifyDeposit()` function performs an unnecessary validator exit check that causes a temporary denial of service whenever the first pending deposit belongs to an exiting validator, preventing deposit verification until the queue progresses or different parameters are used.

The function contains an unnecessary check where it verifies that the validator of the first pending deposit is not exiting.

contracts/strategies/NativeStaking/CompoundingValidatorManager.sol::verifyDeposit()

```
// If the deposit queue is not empty
if (!isDepositQueueEmpty) {
    // Get the parent beacon block root of the next block which is
    // the block root of the validator verification slot.
    bytes32 validatorBlockRoot = BeaconRoots.parentBlockRoot(
        _calcNextBlockTimestamp(firstDepositValidatorCreatedSlot)
    );

    // Verify the validator of the first pending deposit is not exiting.
    // If it is exiting we can't be sure this deposit has not been postponed in the deposit queue.
    // Hence we can not verify if the strategy's deposit has been processed or not.
    IBeaconProofs(BEACON_PROOFS).verifyValidatorWithdrawable( // @audit Unnecessary check causes DoS
        validatorBlockRoot,
        firstPendingDeposit.validatorIndex,
        firstPendingDeposit.pubKeyHash,
        FAR_FUTURE_EPOCH,
        firstPendingDeposit.withdrawableEpochProof,
        firstPendingDeposit.validatorPubKeyProof
    );
}
```

This check is intended to account for postponed deposits, which are moved to the end of the queue and occur when the validator is exiting. However, the existing `deposit.slot < firstPendingDeposit.slot` check is sufficient to determine if deposits have been processed, regardless of whether the first deposit belongs to an exiting validator or not as long as the strategy's validator is not exiting. The additional validator exit verification serves no functional purpose and creates an unnecessary failure condition.

This issue is classified as low impact because it causes temporary service disruption of the deposit verification functionality. While no funds are at risk and workarounds exist (using a different slot with a different first deposit), it prevents legitimate verification operations during realistic operational scenarios.

The likelihood is assessed as medium since the vulnerability occurs whenever there are pending deposits and the first deposit belongs to an exiting validator. This is a realistic operational scenario that can happen during normal staking operations.

Recommendations

Remove the unnecessary validator exit check from the `verifyDeposit()` function. The existing slot-based verification logic is sufficient to determine if deposits have been processed, regardless of the first deposit validator's exit status.

Resolution

The development team has removed the unnecessary validator exit check from the `verifyDeposit()` function.

This issue has been resolved in PR #2640.

ORGN-11	Missing Bounds Validation On Validator Index In `verifyValidator()`
Asset	`contracts/beacon/BeaconProofsLib.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The `verifyValidator()` function in `BeaconProofsLib` lacks bounds validation on the `validatorIndex` parameter, accepting the full `uint64` range instead of constraining it to the beacon chain specification limit of $2^{40} - 1$.

The `verifyValidator()` function accepts a `uint64 validatorIndex` parameter but does not validate that this index is within the valid range of validator indices ($< 2^{40} - 1$). In the Ethereum Beacon Chain specification, validator indices are constrained to 40 bits, representing a maximum of approximately 1.1 trillion validators. However, the function accepts the full `uint64` range, allowing indices up to $2^{64} - 1$.

contracts/beacon/BeaconProofsLib.sol::verifyValidator()

```
function verifyValidator(
    bytes32 beaconBlockRoot,
    bytes32 pubKeyHash,
    bytes calldata proof,
    uint64 validatorIndex, // @audit Missing bounds validation - should be < 2**40
    address withdrawalAddress
) internal view {
    require(beaconBlockRoot != bytes32(0), "Invalid block root");

    // BeaconBlock.state.validators[validatorIndex]
    uint256 generalizedIndex = concatGenIndices(
        VALIDATORS_CONTAINER_GENERALIZED_INDEX,
        VALIDATORS_LIST_HEIGHT,
        validatorIndex // @audit Out-of-bounds index passed without validation
    );
    // ... rest of function
}
```

When `validatorIndex >= 2**40`, the value is passed to `concatGenIndices()` which performs bitwise operations without bounds checking:

contracts/beacon/BeaconProofsLib.sol::concatGenIndices()

```
function concatGenIndices(
    uint256 genIndex,
    uint256 height,
    uint256 index
) internal pure returns (uint256) {
    return (genIndex << height) | index; // @audit No bounds checking on index
}
```

While this represents missing input validation that should be addressed for defensive programming practices, the out-of-bounds validator index does not create exploitable security issues. The merkle proof verification process has natural constraints that prevent the theoretical bit contamination from affecting proof verification:

1. Merkle proofs operate with a fixed depth that bounds which bits of the generalized index are actually consulted

during verification

2. The proof verification process only examines bits within the proof depth, effectively ignoring any out-of-bounds bits
3. Merkle proof verification is inherently constrained by the tree structure rather than the full generalized index space

This issue has an informational severity rating because it represents a code quality issue rather than a security vulnerability. The missing bounds validation violates defensive programming principles but does not enable practical attacks due to the natural constraints of merkle proof verification. The impact is limited to storing and emitting incorrect validator indices in contract state and events.

Recommendations

Reduce the `validatorIndex` parameter from a `uint64` to a `uint40` to ensure it remains within the valid beacon chain specification.

To protect against general malicious input for generalized index calculation, add a bounds validation check to the `concatGenIndices()` function that restricts the size of bits for `index` to the provided `height`.

Resolution

The development team has reduced the `validatorIndex` parameter from a `uint64` to a `uint40` to ensure it remains within the valid beacon chain specification.

This issue has been resolved in PR #2642.

ORGN-12	Miscellaneous General Comments
Asset	All contracts
Status	**Closed:** See Resolution
Rating	Informational

Description

This section details miscellaneous findings discovered by the testing team that do not have direct security implications:

1. **Missing Zero Address Validation In Constructor**

   Related Asset(s): CompoundingValidatorManager.sol

   The constructor does not validate that address parameters are non-zero, which could render contract functionality inoperable if zero addresses are provided.

   contracts/strategies/NativeStaking/CompoundingValidatorManager.sol::constructor()

   ```
   constructor(
       address _wethAddress,
       address _vaultAddress,
       address _beaconChainDepositContract,
       address _ssvNetwork,
       address _beaconProofs,
       uint64 _beaconGenesisTimestamp
   ) {
       WETH = _wethAddress;
       BEACON_CHAIN_DEPOSIT_CONTRACT = _beaconChainDepositContract;
       SSV_NETWORK = _ssvNetwork;
       VAULT_ADDRESS = _vaultAddress;
       BEACON_PROOFS = _beaconProofs;
       BEACON_GENESIS_TIMESTAMP = _beaconGenesisTimestamp;

       require(
           block.timestamp > _beaconGenesisTimestamp,
           "Invalid genesis timestamp"
       );
   }
   ```

   Zero addresses would cause calls to these contracts to fail, making the contract unusable and requiring an upgrade to fix.

   Add zero address checks for all address parameters in the constructor.

2. **Non-Standard Gap Declaration**

   Related Asset(s): CompoundingValidatorManager.sol

   The variable `__gap` is storing 50 slots in an array which is more than the upgrade standard of each inherited contract having 50 slots in total, including used slots.

   contracts/strategies/NativeStaking/CompoundingValidatorManager.sol

   ```
   // For future use
   uint256[50] private __gap;
   ```

   The `CompoundingValidatorManager.sol` contract is currently using 12 storage slots. Which would leave 38 slots to be stored in the `__gap` array.

   Update the `__gap` array to store 38 slots instead of 50.

3. **Documentation and Naming Inconsistencies**

   Related Asset(s): CompoundingValidatorManager.sol

   Several documentation and naming inconsistencies were identified that could lead to developer confusion:

   - The `DepositData` struct holds the amount of ETH in gwei that was deposited, but the comment incorrectly states the variable is holding the amount of ETH in wei.

     contracts/strategies/NativeStaking/CompoundingValidatorManager.sol::DepositData

     ```
     /// @param amountWei Amount of ETH in wei that has been deposited to the beacon chain deposit contract
     ```

   - `ValidatorState.REMOVED` contains a reference to "EigenPod" in the context of this compounding staking strategy, which is terminology from EigenLayer rather than the current system.

     contracts/strategies/NativeStaking/CompoundingValidatorManager.sol::ValidatorState

     ```
     REMOVED, // validator has funds withdrawn to the EigenPod and is removed from the SSV
     ```

   - The comment for `BALANCES_CONTAINER_GENERALIZED_INDEX` incorrectly states that balances are at index 13, when they are actually at index 12. The resulting calculation of 716 is correct despite the typo.

     contracts/beacon/BeaconProofsLib.sol

     ```
     /// Beacon block container: height 3, state at at index 3
     /// Beacon state container: height 6, balances at index 13
     /// (2 ^ 3 + 3) * 2 ^ 6 + 13 = 716
     uint256 internal constant BALANCES_CONTAINER_GENERALIZED_INDEX = 716;
     ```

   Update the comments to reflect the correct information.

4. **Gas Optimization: Missing Early Exit Check for Zero Deposit Slot**

   Related Asset(s): CompoundingValidatorManager.sol

   The deposit verification logic performs expensive proof verification operations before checking if `firstPendingDeposit.slot` is zero. This results in unnecessary gas consumption when the deposit slot is zero, which can occur for validators switching to compounding withdrawal credentials.

   contracts/strategies/NativeStaking/CompoundingValidatorManager.sol::verifyDeposit()

   ```
   // The deposit slot can be zero for validators consolidating to a compounding validator or 0x01 validator
   // being promoted to a compounding one. Reference:
   // ... snip ...
   // We can not guarantee that the deposit has been processed in that case.
   // solhint-enable max-line-length
   require(
       deposit.slot < firstPendingDeposit.slot || isDepositQueueEmpty,
       "Deposit likely not processed"
   );
   ```

   Users may waste gas on proof verification operations when `firstPendingDeposit.slot` is zero, as the proof verification will be performed before the slot comparison check that would have failed anyway.

   Add an early exit check at the beginning of the function to verify that `firstPendingDeposit.slot` is not zero before proceeding with expensive proof verification operations:

   contracts/strategies/NativeStaking/CompoundingValidatorManager.sol::verifyDeposit()

   ```
   require(firstPendingDeposit.slot != 0, "First pending deposit slot cannot be zero");
   ```

   This optimization will save gas by avoiding unnecessary computations in cases where the deposit verification is guaranteed to fail due to a zero slot value.

Recommendations

Ensure that the comments are understood and acknowledged, and consider implementing the suggestions above.

Resolution

The development team's responses to the raised issues above are as follows:

1. **Missing Zero Address Validation In Constructor**: *"Acknowledged. Zero address checks came from an early issue in `web3.js` that is now fixed. We've chosen to not implement this fix to save on contract bytecode size."*
2. **Non-Standard Gap Declaration**: *"Fixed. The `__gap` array has been updated to store 40 slots instead of 50, accounting for the 10 slots that are currently used."*
3. **Documentation and Naming Inconsistencies**: *"Fixed. The comments have been updated to reflect the correct information."*
4. **Gas Optimization: Missing Early Exit Check for Zero Deposit Slot**: *"Fixed. An early exit check has been added to the function to avoid unnecessary gas consumption when the deposit slot is zero."*

The fixes above have been implemented in PRs #2657 and #2663.

Appendix A Vulnerability Severity Classification

This security review classifies vulnerabilities based on their potential impact and likelihood of occurance. The total severity of a vulnerability is derived from these two metrics based on the following matrix.

Impact	High	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Low	Low	Medium
		Low	Medium	High
			Likelihood	

Table 1: Severity Matrix - How the severity of a vulnerability is given based on the *impact* and the *likelihood* of a vulnerability.

References

[1] Sigma Prime. Solidity Security. Blog, 2018, Available: `https://blog.sigmaprime.io/solidity-security.html`. [Accessed 2018].

[2] NCC Group. DASP - Top 10. Website, 2018, Available: `http://www.dasp.co/`. [Accessed 2018].

σ'