BAIL security

Hyperdrive
LST

FINAL REPORT

April ‘2025

Disclaimer:

Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

The content of this assessment is not an investment. The information provided in this report is for general informational purposes only and is not intended as investment, legal, financial, regulatory, or tax advice. The report is based on a limited review of the materials and documentation provided at the time of the audit, and the audit results may not be complete or identify all possible vulnerabilities or issues. The audit is provided on an "as-is," "where-is," and "as-available" basis, and the use of blockchain technology is subject to unknown risks and flaws.

The audit does not constitute an endorsement of any particular project or team, and we make no warranties, expressed or implied, regarding the accuracy, reliability, completeness, or availability of the report, its content, or any associated services or products. We disclaim all warranties, including the implied warranties of merchantability, fitness for a particular purpose, and non-infringement.

We assume no responsibility for any product or service advertised or offered by a third party through the report, any open-source or third-party software, code, libraries, materials, or information linked to, called by, referenced by, or accessible through the report, its content, and the related services and products. We will not be liable for any loss or damages incurred as a result of the use or reliance on the audit report or the smart contract.

The contract owner is responsible for making their own decisions based on the audit report and should seek additional professional advice if needed. The audit firm or individual assumes no liability for any loss or damages incurred as a result of the use or reliance on the audit report or the smart contract. The contract owner agrees to indemnify and hold harmless the audit firm or individual from any and all claims, damages, expenses, or liabilities arising from the use or reliance on the audit report or the smart contract.

By engaging in a smart contract audit, the contract owner acknowledges and agrees to the terms of this disclaimer.

Due to the large volume of H/M issues found during the second round,
a complete new audit round is required. Bailsec does not consider this scope deployment ready.
Furthermore,substantial refactoring in the resolution round was applied which was not validated by Bailsec

1. Project Details

Important:
Please ensure that the deployed contract matches the source-code of the last commit hash.

Project	Hyperdrive - LST
Website	hyperdrive.fi
Language	Solidity
Methods	Manual Analysis
Github repository	Round 1 https://github.com/ambitfi/hyperdrive-contracts/tree/6fefd66809e3b751d2f3518dbfa626fe4ae6aed6/packages/staking/contracts/protocol Round 2 https://github.com/ambitfi/hyperdrive-contracts/tree/0befa1b188388ee7f41580ed3ee2beccfee2def3/packages/staking/contracts/protocol
Resolution 1	https://github.com/ambitfi/hyperdrive-contracts/tree/9e3127eae0feb802d109c613bdc9e415e3efd914/packages/staking/contracts/protocol

Due to the large volume of H/M issues found during the second round,
a complete new audit round is required. Bailsec does not consider this scope deployment ready.
Furthermore,substantial refactoring in the resolution round was applied which was not validated by Bailsec

2. Detection Overview

Severity	Found	Resolved	Partially Resolved	Acknowledged (no change made)	Failed Resolution
High	13	8		2	
Medium	6	4		2	
Low	13	6		5	
Informational	15	6		7	
Governance	1			1	
Total	48	24		17	

2.1 Detection Definitions

Severity	Description
High	The problem poses a significant threat to the confidentiality of a considerable number of users' sensitive data. It also has the potential to cause severe damage to the client's reputation or result in substantial financial losses for both the client and the affected users.
Medium	While medium-level vulnerabilities may not be easy to exploit, they can still have a major impact on the execution of a smart contract. For instance, they may allow public access to critical functions, which could lead to serious consequences.
Low	Poses a very low-level risk to the project or users. Nevertheless, the issue should be fixed immediately.
Informational	Effects are small and do not pose an immediate danger to the project or users.
Governance	Governance privileges which can directly result in a loss of funds or other potential undesired behavior.

Due to the large volume of H/M issues found during the second round,
a complete new audit round is required. Bailsec does not consider this scope deployment ready.
Furthermore,substantial refactoring in the resolution round was applied which was not validated by Bailsec

3. Detection

Round 1

The first round was conducted by Marc Weiss as an independent solo researcher joining Bailsec's audit process.

Issue_01	Assets are treated as shares
Severity	**High**
Description	Inside the redeem function, shares are sent by the controller and assets are withdrawn by the receiver. In this case, after the asset calculation in `previewRedeem`, the wrong order of parameters is being forwarded inside claimableWithdraw, as assets are where shares should be, and the other way around: *require(* *shares <= $.requests[controller].claimableShares,* *ERC4626ExceededMaxRedeem(controller, shares, $.requests[controller].claimableShares));* *assets = previewRedeem(controller, shares);* *claimableWithdraw(msg.sender, receiver, controller, assets, shares);* Therefore, shares will actually be sent to the receiver instead of assets and assets will be subtracted from the shares mapping: *payable(receiver).sendValue(assets);* *$.requests[controller].claimableShares -= shares;* *$.requests[controller].claimableAssets -= assets;*

Due to the large volume of H/M issues found during the second round, a complete new audit round is required. Bailsec does not consider this scope deployment ready. Furthermore,substantial refactoring in the resolution round was applied which was not validated by Bailsec

Recommendations	Consider changing shares for assets: *require(* *shares <= $.requests[controller].claimableShares,* *ERC4626ExceededMaxRedeem(controller, shares, $.requests[controller].claimableShares));* *assets = previewRedeem(controller, shares);* *- claimableWithdraw(msg.sender, receiver, controller, assets, shares);* *+ claimableWithdraw(msg.sender, receiver, controller, shares, assets);*
Comments / Resolution	Fixed at commit: 04d792ba8bdf969a9aa7b3a414d55085aa756bba

Issue_02	Shares are drainable as soon as they become claimable
Severity	High
Description	Anyone can drain all the claimable shares from all the controllers as there is no requirements in regards to who the caller is, and whether they are owners or operators: *function withdraw(* *uint256 assets,* *address receiver,* *address controller* *) external override(ERC7535Upgradable, IERC7535) returns (uint256 shares) {* *StakingStorage storage $ = getStakingStorage();* *require(* *assets <= $.requests[controller].claimableAssets,* *ERC4626ExceededMaxWithdraw(controller, assets, $.requests[controller].claimableAssets)* *);* *shares = previewWithdraw(controller, assets);* *claimableWithdraw(msg.sender, receiver, controller, shares, assets);* *}* This allows anyone to pass a valid controller address where they claimable assets and shares and set themselves as the receiver, which would allow to drain the funds from such controllers: *payable(receiver).sendValue(assets);* *$.requests[controller].claimableShares -= shares;* *$.requests[controller].claimableAssets -= assets;* *$.totalClaimableAssets -= assets;*

Recommendations	Consider adding the onlyOwnerOrOperator modifier to the redeem and withdraw functions
Comments / Resolution	Fixed at commit: dd490f4f130ba280e2510fa507b2c90f27e1d1dd

Issue_03	Receivers can grief senders on withdrawals
Severity	**Medium**
Description	When a controller calls redeem or withdraw for a receiver, such receiver receives the HYPE withdrawn via a external call to their address where they can grief the controller and load a lot of data into memory so that the controller needs to pay close to the gas limit to process the transaction. *function claimableWithdraw(* *address caller,* *address receiver,* *address controller,* *uint256 shares,* *uint256 assets* *) private {* *StakingStorage storage $ = getStakingStorage();* *payable(receiver).sendValue(assets);* *$.requests[controller].claimableShares -= shares;* *$.requests[controller].claimableAssets -= assets;* *$.totalClaimableAssets -= assets;*
Recommendations	Use the version of `call` that does prevent gas bomb attacks and limit the amount of gas that can be spent by the receiver. ```diff function claimableWithdraw(address caller, address receiver, address controller, uint256 shares,

	uint256 assets) private { StakingStorage storage $ = getStakingStorage(); - payable(receiver).sendValue(assets); + assembly { + success := call(gasLimit, receiver, amount, 0, 0, 0, 0) + } $.requests[controller].claimableShares -= shares; $.requests[controller].claimableAssets -= assets; $.totalClaimableAssets -= assets; ```
Comments / Resolution	Acknowledged

Issue_04	Unfair distribution of delegated amounts depending on the order of validators
Severity	**Low**
Description	On delegations, the function loops through all the active validators and calculates the percentage respective the total weight. Then, the delegation amounts are distributed. As seen in the code below, the last validator of the list of validators (length - 1), will receive more delegation in comparison to other validators as it will always sweep all the dust: *uint256 percentage = (configuration.targetWeighting * 1e18) / totalTargetWeighting;* *// for the last validator we transfer the remaining in the case where there is dust due to rounding* *Wei delegation = i == length - 1* *? WeiMath.toWei(assets) - delegated* *: WeiMath.toWei((assets * percentage) / 1e18);* *delegate(validator, delegation);* If the list of validators does not change in a long enough time frame and the same validator is always the last on in the list, the distribution of delegated amounts is unfair to all the other validators
Recommendations	There could be a variable that tracked the last validator that got benefited from the dust and be updated so next time that validator + 1 gets the dust delegated until it reaches the max length of the array of validators, where it would start again.
Comments / Resolution	The developer team has stated that it is accepted as the additional complexity is not worth it for the sake of 1 HYPE of dust

Issue_05	Missing input validation
Severity	**Low**
Description	In the initialize function in the vault, check that depositLockUp is bigger than 0: *function initialize(* *IAddressRegistry registry,* *string memory name,* *string memory symbol,* *uint64 _tokenIndex,* *uint256 initialShares,* *uint256 depositLockUp* *) public initializer {* *__ERC20_init_unchained(name, symbol);* *__UUPSUpgradeable_init_unchained();* *__AdminAccessControl_init_unchained(registry);* *__Upgradeable_init_unchained();* *__RoleBasedAccessControl_init_unchained();* *StakingStorage storage $ = getStakingStorage();* *$.tokenIndex = _tokenIndex;* *$.depositLockUp = depositLockUp;* *if (initialShares > 0) {* *_mint(address(this), initialShares);* *}* *}* - In all major entry points such as `deposit` check that the specified receiver address is not address 0 - In `WithdrawQueueUpgradeable` make sure to validated the address of the stakingVault not to be 0: *function initialize(IAddressRegistry registry, IStakingVault stakingVault) public initializer {*

	__UUPSUpgradeable_init_unchained(); *__AdminAccessControl_init_unchained(registry);* *__Upgradeable_init_unchained();* *WithdrawQueueStorage storage $ = getWithdrawQueueStorage();* *$.stakingVault = stakingVault;* *}*
Recommendations	Add the previously mentioned validations
Comments / Resolution	Fixed at commit: 989213eb0b0e702f4850c11a7a9ad918b75cfaab

Issue_06	Redundant sstore when `depositAccounting.blockNumber = block.number`
Severity	Informational
Description	When a deposit gets tracked after depositing in the vault, there are two cases, when `depositAccounting.blockNumber != block.number` and when `depositAccounting.blockNumber = block.number`. In the first case, it resets the crediting, and set the new block.number to the current one. But, in the case where `depositAccounting.blockNumber = block.number`, it does set again the block.number to the same one than the current one: *function trackDeposit(address proxy, uint256 amount) private {* *StakingStorage storage $ = getStakingStorage();* *DepositAccounting storage depositAccounting = $.depositAccounting[proxy];* *if (depositAccounting.blockNumber != block.number) {* *depositAccounting.crediting = 0;* *}* *depositAccounting.blockNumber = block.number;* *depositAccounting.crediting += amount;* *}*
Recommendations	Only update the `depositAccounting.blockNumber` if it is not equal to the current block.number ```diff function trackDeposit(address proxy, uint256 amount) private { StakingStorage storage $ = getStakingStorage(); DepositAccounting storage depositAccounting = $.depositAccounting[proxy];

	if (depositAccounting.blockNumber != block.number) { depositAccounting.crediting = 0; + depositAccounting.blockNumber = block.number; } - depositAccounting.blockNumber = block.number; depositAccounting.crediting += amount; }
Comments / Resolution	Fixed at commit: 0befa1b188388ee7f41580ed3ee2beccfee2def3

Round 2

Due to the large volume of H/M issues found during the second round, a complete new audit round is required. Bailsec does not consider this scope deployment ready. Furthermore,substantial refactoring in the resolution round was applied which was not validated by Bailsec

This is the second round which was conducted on a commit which already included the fixes for issues which were found during the first round. It was conducted by Charles Wang.

Blackbox Assumptions

The architecture triggers various write and read interactions b/w Hyperdrive L1 and Hyperdrive EVM chain. There are several assumptions which cannot be verified, hence we rely on these assumptions to work as expected. If under any case, these assumptions do not hold, the codebase will become highly insecure.

- totalAssets includes at all times the total HYPE amount in the system and covers all edge-cases
- All precompile calls work as expected, Bailsec has knowledge over function selector correctness
- L1; EVM blocks are sequential
- EVM reads from previous L1 block, precompile call during EVM block does not alter L1Read
- L1 changes are reflected in the subsequent EVM block and accessed via L1Read
- No “on behalf” delegation for proxies on L1 allowed
- On EVM, Hype has 18 decimals. On L1, 8 decimals

Vault

ERC7535Upgradeable

The ERC7535Upgradeable contract forms the parent contract for the StakingVault and exposes various functions which are mostly overridden. It basically serves as the modified ERC4646 base

Privileged Functions

- none

Issue_07	Incorrect asset name
Severity	**Low**
Description	The asset is defined as ETH but is in reality HYPE. Moreover the child contract already exposes a HYPE variable. This can be confusing for third parties.
Recommendations	Consider handling this scenario properly.
Comments / Resolution	Acknowledged.

StakingVaultUpgradeable

The following changes have been identified as logical changes/OOS changes/intrusive changes that warrant an additional audit round; however, they may not account for all such changes introduced:

- **implementation of refund mechanism within mint**
- **previewUndelegate logic (minimal but must be audited in full callpath)**
- **undelegate logic**
- **finalizeRequestRedeem logic**
- **finalizeWithdraw logic (minimal but must be audited in full callpath)**

The StakingVaultUpgradeable contract is a heavily modified ERC4626 vault with asynchronous withdrawals based on a FiFo queue. Users can provide native HYPE to receive shares and these HYPE are bridged to the L1 and delegated to specified validators. Withdrawals are possible via a redemption flow, starting via the requestRedeem function which transfers shares into the vault and pushes the request in the pendingQueue to be undelegated at some point in the future. As long as it has not been undelegated it can always be cancelled via the cancelRequest function and even if it has been undelegated it can still be cancelled via the forfeitRedeemRequest function.

Appendix: Delegation Flow

Upon each deposit/mint, the delegate function is invoked. The flow for delegating funds from the StakingVault to validators works as follows:

a) The totalTargetWeighting amount is fetched which is the aggregate of the individual targetWeights from all validators

b) A loop over all validators is executed and the percentage amount from each validator based on the totalTargetWeighting amount is fetched:

> percentage1e18 = validator.targetWeighting * 1e18 / totalTargetWeighting

c) The amount is split based on the percentage and delegated to each validator proportionally:

> amount = assets * percentage1e18 / 1e18

d) For the last validator, simply the leftover amount is delegated to ensure no dust remains in the contract:

> leftover = assets - delegated

The detailed fund flow is as follows:

For each validator:

a) HYPE is transferred from StakingVault to the active proxy

b) HYPE is bridged to L1 spot balance for active proxy

c) HYPE is transferred from L1 spot balance to undelegated balance

d) HYPE is transferred from undelegated balance to desired validator

The execution is written to the L1 state and visible in the next block only.

Appendix: totalAssets

As with all ERC4626 vaults (and/or modified versions), the totalAssets function is fundamental to calculate the ER of the vault, which is used by most functionalities. In a standard vault, the totalAssets function simply returns the current balance of the vault. However, for any modified vault, additional measurements may become necessary. This is the case for the LST vault as assets are not simply sitting in the vault but are spread among a few different spots. These spots will be iterated below:

a) Vault balance: Any HYPE balance which is sitting in the vault. Under normal circumstances, the vault does not have any balance besides the claimable balance.

b) DepositAccounting: Any HYPE balance which was bridged to the L1 during the current block

c) Proxy balance (EVM): Any proxy balance on the EVM which is received by withdrawing from the L1 spot balance

d) Proxy balance (L1): Any proxy balance on the L1 which is received after the 7 day transition phase during the undelegation but not yet bridged to the EVM

e) Delegated: Any proxy balance which is delegated to a validator

f) Undelegated: Any proxy balance which was undelegated from a validator. During the normal business logic, this should never happen as an undelegation immediately pushes the delegated balance into the totalPendingWithdrawal state

g) totalPendingWithdrawal: Any proxy balance which was undelegated less than 7 days ago sits temporarily in this state before being moved to the L1 proxy balance

Appendix: Redemption Flow

Users can redeem shares and claim them once the request has been finalized, this happens in the following three steps:

a) Redemption intent: A user requests a redemption via the requestRedeem function. The request is then pushed into the pendingQueue

b) Pending processing: Once the request has reached the first spot in the queue, it will be processed via an undelegation. It is then pushed into the withdrawQueue.

c) Withdraw processing: Once a request has been pushed into the withdrawQueue and has reached the first spot in the queue, it can be finalized as soon as the available balance on the EVM is sufficient to honor the request.

d) Claiming: Once a request has been finalized, it can be claimed by the controller or operator.

Appendix: previewUndelegate function

The previewUndelegate function is a fundamental part in the undelegation process. The core of this function is to calculate the amount which should be withdrawn from each validator based on the provided _wei input parameter and a validator's share on the total delegated amount (unlocked).

If for example the amount is 100e18 and we have 10 unlocked validators where each validator has a delegation of 100e18, this means the total delegated (unlocked) amount is 1000e18. Thus each validator owns a 10% share of the total amount and based on this percentage share, the undelegation of the 100e18 amount should happen. The result is that 10e18 will be undelegated from each unlocked validator. Below we will describe the mechanism of how that is calculated in-depth:

a) A loop over all proxies is executed

b) The delegations for the proxy and all its corresponding validators is fetched from the system contract via:

   DELEGATIONS_PRECOMPILE_ADDRESS.staticcall(abi.encode(user))

c) A loop over all (unlocked) validators is made while the undelegation[] array keeps track of the sum for each individual validator and the total variable keeps track of the whole aggregated sum of all undelegations

d) After this has been done for each proxy address and the total variable is indeed the aggregated undelegatable balance from all unlocked validators, another loop is being executed which loops over all undelegations, calculates the proportional rate of each validator's undelegatable amount based on the total undelegatable amount and then calculates the amount to be undelegated from each validator based on the proportional share:

   Calculate validator's percentage in 1e18 based on the total undelegatable amount:

   > percentageIn1e18 = undelegation[i] * 1e18 / total (truncated)

   Calculate the amount to undelegate from each validator based on the target amount and the percentage in 1e18:

   > _wei * percentageIn1e18 / 1e18 (rounded up)

Core Invariants:

INV 1: Within undelegate, sum of undelegations[] should always be larger than _wei

INV 2: Delegate must delegate the full idle amount deducted by the claimable assets

INV 3: Request can only be cancelled in PENDING state

INV 4: Requests can only be forfeited in WITHDRAWING state

INV 5: The requestRedeem function must be exchange rate neutral

INV 6: The vault can only accumulate idle funds in case a request is forfeited

INV 7: The prepareRequestRedeem function must withdraw all L1 spot balances

INV 8: The undelegate function must be called in the same tx as previewUndelegate

INV 9: Within the deposit function, asset parameter must match msg.value

INV 10: The undelegate function must only be callable by the WithdrawQueue contract

INV 11: The totalClaimableAssets variable must reflect how much assets can be claimed by fulfilled requests

INV 12: Within the previewUndelegate function, the undelegate[] array must include all validators, including empty ones

Privileged Functions

- setWithdrawQueue
- grantRole
- revokeRole
- addProxy
- addValidator

Issue_08	Governance Issue: Full governance control
Severity	**Governance**
Description	Currently, governance of this contract has several privileges for invoking certain functions that can drastically alter the contracts behavior: - updating proxy implementation - changing withdrawQueue - adding invalid validators - adding invalid proxy contracts - ....
Recommendations	Consider incorporating a Gnosis Multisignature contract as owner and ensuring that the Gnosis participants are trusted entities.
Comments / Resolution	Acknowledged.

Issue_9	First depositor can mint shares for free
Severity	High
Description	Currently, it is expected that an initial amount of shares is minted to the contract without backing. This will result in a state where totalSupply != 0 and totalAssets = 0. In itself, this is already a faulty state where most vaults are not compatible with, as this is never a state which would be naturally reached. This can be exploited by the first depositor when calling the mint function as it naturally does not require any assets to be provided due to the way how the asset amount is computed: > assets = shares * totalAssets / totalSupply Due to the fact that totalAssets is zero, the required amount of assets is zero and the first depositors will always get free shares until a user deposits via the deposit function.
Recommendations	Consider removing the initial mint.
Comments / Resolution	Acknowledged, during the initialize function no assets are being forced to be transferred in, this should be handled in case initialShares > 0. Otherwise it is simply sufficient to set initialShares parameter to zero.

Issue_10	Several ways allow for DoS'ing the queue
Severity	**High**
Description	The contract exposes a FiFo queue which is handled within the WithdrawQueueUpgradeable contract. It is possible to trivial DoS this queue via at least three scenarios: a) Create a large amount of small requests to congest the pendingQueue and withdrawQueue b) Create a large amount of normal or small requests and cancel them again to provoke the early return within the processPendingQueue function in the WithdrawQueueUpgradeable contract c) Create a large amount of normal or small requests, wait until they are processed and forfeit them (if fixed) to congest the pendingQueue and withdrawQueue. The root-cause for all of these lies within the fact that there are no cancellation fees and a lack of minimum enqueue amount.
Recommendations	Consider implementing redemption fees for enqueueing as well as a minimum enqueue amount.
Comments / Resolution	The withdrawal/undelegation flow has been fully refactored. This must be fully audited and formally verified with all possible edge-cases in order to mark as resolved in an effort to ensure no side-effects are introduced due to the new implementation. This status is subject to update.

Issue_11	Contract is drainable due to incorrect totalAssets value
Severity	High
Description	The totalAssets function is the most fundamental function in all vaults as it determines the ER by returning the AUM value. In the current implementation, it adds the full native balance towards the AUM value: *total += address(this).balance;* This is incorrect, as the totalClaimableBalance must be deducted. A malicious user can abuse this by redeeming a large amount such that the corresponding shares are burned and totalClaimableAssets is increased but not reflected in totalAssets which will increase the ER by a large degree, followed by subsequent redemptions to exploit the increased ER.
Recommendations	Consider deducting totalClaimableAssets.
Comments / Resolution	Resolved.

Issue_12	Claims can be fully DoS'd due to incorrect delegation amount
Severity	**High**
Description	The totalClaimableAssets value is responsible for ensuring that users can indeed claim their assets from the vault. It basically acts as a "safe reserve". However, this is not incorporate into the delegate function as simply the whole native balance is delegated upon each deposit: *delegate{address(this).balance}.fromWei();* This means that no claims are ever possible because no safe reserve is left for users to claim.
Recommendations	Consider not delegating the full native balance.
Comments / Resolution	Resolved.

Issue_13	Locked funds in EVM proxy due to undelegation/withdrawing flow
Severity	High
Description	The undelegation flow is described in the corresponding appendix and one specific part is the fact that the full entry.assets amount is being undelegated: *if (tryUndelegate(entry.assets))* Once a request has been undelegated it is pushed into the withdrawQueue and can be finalized as soon as the withdrawable balance on the EVM (proxy balances + StakingVault balance) is sufficient to honor the request. The finalizeWithdraw function only transfers the necessary funds from the proxy: *assets -= Math.min(assets, balance);* This means any leftover funds in the proxy contract remain simply locked as they are never claimed nor compounded back. There are two distinct scenarios where this applies: - In case the asset amount during the finalizeRequestRedeem function is lower than the undelegated asset amount - In case there are funds in the StakingVault, for example during forfeiting
Recommendations	A straightforward recommendation would be to simply withdraw all funds from a proxy. However, doing that would mean the potential withdrawing of funds batched from other requests as well while only increasing totalClaimableAssets for the current request. This means these funds will be compounded back, essentially breaking the redemption process. Thus we recommend simply claiming these funds from the proxy

	and eventually depositing them back, this needs increased flexibility in the fallback function.
Comments / Resolution	The withdrawal/undelegation flow has been fully refactored. This must be fully audited and formally verified with all possible edge-cases in order to mark as resolved in an effort to ensure no side-effects are introduced due to the new implementation. This status is subject to update.

Issue_14	First depositor will donate assets to vault initiator
Severity	**High**
Description	Currently, it is expected that an initial amount of shares is minted to the contract without backing. This will result in a state where totalSupply != 0 and totalAssets = 0. In itself, this is already a faulty state where most vaults are not compatible with, as this is never a state which would be naturally reached. If now the first deposit is executed, a part of the assets are simply donated to the unbacked shares, resulting in a loss for the first depositor. **Illustrated**: Contract is deployed and 1e18 shares are minted initially - totalSupply = 1e18 - totalAssets = 0 - Alice calls deposit with 1e18 assets - convertToShares - Alice will receive 1e18 shares - Since the total supply is now 2e18 assets, Alice will only own 50% of her initial deposit In the example one can see that Alice unexpectedly donated assets to the unbacked shares and experienced a partial loss of funds.

	This issue can be inflated if the initial mint amount is higher, it
Recommendations	Consider simply transferring the corresponding asset amount in during the initialization to ensure the share amount is backed and to avoid this faulty state.
Comments / Resolution	Acknowledged.

Issue_15	Validators cannot be updated or removed
Severity	**High**
Description	Within the ValidatorSetLib, the update function allows for updating and disabling validators. However, this function is uncallable due to the missing implementation within the StakingVault. This can result in large issues, specifically if for example one validator is disabled on the L1. Deposits would now always revert because the weights cannot be updated.
Recommendations	Consider implementing this functionality.
Comments / Resolution	Resolved, the update function is now callable which allows setting the targetWeighting to zero which then skips the loop within delegate.

Issue_16	Executions will increase vault exchange rate
Severity	High
Description	In the corresponding Appendix we have already elaborated the asynchronous withdrawal mechanism where a redemption is first requested and then executed. The user will then receive the corresponding asset amount for the provided share amount either based on the time of the request intent or at the time of the first/final execution, picking the lower of these values. In the scenario where the exchange rate was lower at the time of the request intent this means the user will get less assets for his shares than he would receive based on the current exchange rate. This redemption will inherently increase the exchange rate and grant all users in the vault more assets for their corresponding shares. This issue can be abused by smart users who are observing the current pending execution requests with such a state. Users can simply deposit, call execute to increase the exchange rate and are now immediately in profit with their position.
Recommendations	Consider simply using the current exchange rate at the time of the request execution.
Comments / Resolution	The withdrawal/undelegation flow has been fully refactored. This must be fully audited and formally verified with all possible edge-cases in order to mark as resolved in an effort to ensure no side-effects are introduced due to the new implementation. This status is subject to update.

Issue_17	Forfeit is wrongly implemented
Severity	**High**
Description	The forfeitRequestRedeem function is meant to have a similar functionality as the cancelRequestRedeem function by simply refunding a user's shares once the request is in the WITHDRAWING state. However, currently these assets are just donated to the vault due to a blunder in the code.
Recommendations	Consider implementing a proper refund.
Comments / Resolution	Resolved, the whole forfeiting logic has been removed.

Issue_18	Funds can be fully removed from the delegation system via forfeitRedeemRequest
Severity	**High**
Description	If the forfeitRedeemRequest function is fixed and properly refunds shares to a user, it can be trivially abused to undelegate funds from validators via executing requests and forfeiting them in a repetitive manner by requesting, forfeiting, reclaiming shares and repeating. All requested funds will then sit in the air for 7 days. This can be done to fully remove funds from all validators in a few blocks if the queue is currently empty.
Recommendations	Consider simply removing this function.
Comments / Resolution	Resolved, the whole forfeiting logic has been removed.

Issue_19	Loss of excess msg.value during mint
Severity	**Medium**
Description	Currently, the mint function allows users to provide a shares parameter and a corresponding msg.value. It then expects the *msg.value >= assets* to fulfill the share creation request. In itself, this logic is correct. However, there is currently no refund to a user if a larger msg.value than necessary is provided which means that these funds are essentially donated to the vault. Furthermore, within the second mint function, the maxAmount in check is completely redundant as the msg.value is the only source of truth.
Recommendations	Consider refunding any excess provided msg.value. Extensive testing and potential side-effects must be considered.
Comments / Resolution	Resolved, a refund mechanism has been incorporated. While this refund mechanism is properly implemented by simply returning msg.value - assets, it is recommended to re-audit the mint flow in an effort to ensure no side-effects can occur.

Issue_20	Dangerous down-sizing within previewWithdraw
Severity	**Low**
Description	The previewWithdraw function sizes down the necessary amount of provided shares as follows: *shares = Math.min(shares, $.requests[controller].claimableShares);* This is towards the favor of the user since it can result in decreasing the necessary amount of shares for a desired amount of assets. This also means using the withdraw function, users can bypass the rounding, while via the redeem function, users will still experience the unfavored rounding.
Recommendations	Consider removing this downsizing, implementing additional tests and fuzz scenarios for the withdrawal process.
Comments / Resolution	Resolved.

Issue_21	depositLockUp cannot be changed
Severity	Low
Description	The depositLockUp determines the interval in which the getActiveProxy is determined. It is currently not changeable which may result in limitations.
Recommendations	Consider implementing a function which allows changing the depositLockUp.
Comments / Resolution	Resolved, a setDelegationLockUp function has been introduced which allows for setting the delegationLockUp. The depositLockUp has just been renamed without any logical changes.

Issue_22	Truncation during previewUndelegate may result in _wei amount not being reached
Severity	Low
Description	The previewUndelegate function returns how the _wei amount is being split among all validators based on the corresponding proportional validator share to the total amount. The way how this is done is explained in the corresponding appendix. It is theoretically possible for truncation to result in a zero percentage amount (while it should be non-zero), which would then result in the aggregate of all delegations to become smaller than _wei (even if rounded up during the final calculation), which would essentially break the withdrawal process.
Recommendations	Consider implementing additional testing and fuzzing to determine the likelihood of this issue to occur.
Comments / Resolution	The withdrawal/undelegation flow has been fully refactored. This must be fully audited and formally verified with all possible edge-cases in order to mark as resolved in an effort to ensure no side-

	effects are introduced due to the new implementation. This status is subject to update.

Issue_23	No address(0) possibility for withdrawQueue
Severity	**Low**
Description	The setWithdrawQueue function allows for setting the withdrawQueue variable which is initially address(0). Once it is set, it can never be set back to address(0). While it is possible to set it to a custom contract or EOA in order to prevent redemptions, a cleaner way would be to simply setting it to address(0). This increases flexibility for governance to prevent withdrawals once desired.
Recommendations	Consider allowing withdrawQueue to be set to address(0).
Comments / Resolution	Resolved.

Issue_24	Potential OOG due to too many proxies/validators
Severity	**Low**
Description	Several functions loop over the proxy and validator arrays. In the scenario where these arrays become unreasonably large, this loop will consume excessive gas which can result in a function revert due to the “out of gas” issue.
Recommendations	Consider keeping that in mind when adding new proxies and validators. Specifically because these can never be removed.
Comments / Resolution	Acknowledged.

Issue_25	Redundant previewUndelegate call within undelegate
Severity	**Informational**
Description	The previewUndelegate function is called within the undelegate function. This practice is redundant as it is already called in the function before and the result parameters could simply be passed as memory in the calldata.
Recommendations	We do not recommend a change as this mechanism is very complex and it could result in the introduction of new issues.
Comments / Resolution	The withdrawal/undelegation flow has been fully refactored. This must be fully audited and formally verified with all possible edge-cases in order to mark as resolved in an effort to ensure no side-effects are introduced due to the new implementation. This status is subject to update.

Issue_26	Force transfer of native HYPE can increase stuck funds within EVM proxy
Severity	**Informational**
Description	As already elaborated in another issue, funds during the withdrawal process will inadvertently get stuck within the EVM proxy as the withdrawal flow only withdraws in fact how much is necessary. If a forced HYPE transfer is being executed, this will increase address(this).balance which then in turn decreases the withdrawing balance from the EVM proxies during the next execution. This issue is only rated as informational because the vault has essentially only “stuck” funds which are donated anyways.
Recommendations	Consider keeping this issue in mind.
Comments / Resolution	Acknowledged.

Issue_27	Incorrect return value by setOperator
Severity	**Informational**
Description	The setOperator function returns true even if the operator setting was not changed. This can result in misunderstandings.
Recommendations	Consider adjusting the return value.
Comments / Resolution	Acknowledged.

Issue_28	Lack of controller address check within requestRedeem
Severity	Informational
Description	Most of the time, exploits happen due to arbitrary user inputs or users invoking functions which are not meant to be invoked by users, one can argue that a large user flexibility is a great seed for exploits. Therefore, at BailSec, we are of the opinion that codebases should never provide more user flexibility than necessary during the normal business logic. For example, the controller address during requestRedeem can be set to address(0).
Recommendations	Consider implementing a sanity check.
Comments / Resolution	Resolved.

Issue_29	Redundant requestId parameter for claimCancelRedeemRequest
Severity	Informational
Description	The aforementioned function exposes a requestId parameter which must be zero. This is redundant and could be simply removed. Moreover, this function can be called even without any claimable shares.
Recommendations	Consider simply removing this parameter.
Comments / Resolution	Acknowledged.

Issue_30	Unused error
Severity	**Informational**
Description	The contract contains one or more errors which are completely unused. This will increase contract size for no reason and can confuse third-party reviewers: - *error InsufficentLiquidityForRedemption(uint256 expected, uint256 actual);*
Recommendations	Consider reconsidering the usage of this error.
Comments / Resolution	Resolved.

Issue_31	Unnecessary usage of ValidatorConfiguration struct
Severity	**Informational**
Description	During the resolution round, the ValidatorConfiguration struct has been adjusted which now simply reflects the following: *struct ValidatorConfiguration {* *uint8 targetWeighting;* *}* This is unnecessary complex as no struct is needed and simply targetWeighting can be used directly.
Recommendations	Consider simply removing the struct and use targetWeighting.
Comments / Resolution	

Withdrawal Queue

WithdrawQueueLib

The WithdrawQueueLib contract is a simple library contract which is used by the WithdrawQueueUpgradeable and StakingVaultUpgradeable contracts. It exposes the following QueueEntry struct:

a) requestId
b) timestamp
c) controller

Furthermore, it stores corresponding constants for the status

Privileged Functions

- none

Issue_32	Unused variable(s)
Severity	**Informational**
Description	The contract contains one or more unused variables which will confuse third-party reviewers and increases the contract size for no reason. - uint8 public constant FINALIZED = 3;
Recommendations	Consider elaborating the use of this variable.
Comments / Resolution	Resolved.

WithdrawQueueUpgradeable

The following changes have been identified as logical changes/OOS changes/intrusive changes that warrant an additional audit round; however, they may not account for all such changes introduced:

- **new fee application introduced with all setters**
- **undelegatedAt setting newly introduced**
- **tryUndelegate new logic in context to previewUndelegate**
- **implementation of new sweep function**

The WithdrawQueueUpgradeable contract is an important component of the StakingVault as it manages the withdrawal queue for the asynchronous withdrawal process. Users must first request a redemption for an amount of shares. This request will then be enqueued via the WithdrawQueue contract and stored at the end of the pendingQueue. Once it has been removed from the pendingQueue and the corresponding asset amount is undelegated or in the process of undelegation, it is pushed into the withdrawQueue where it can then be finalized as soon as the StakingVault has sufficient assets to honor the request.

Processing of the pendingQueue and withdrawQueue is happening once per block via the execute function. This function is permissionless and incentivizes users to call it as a part of the redeemed asset amount flows as fee towards the processors. The fee module is share based and increments a caller's share by one each execute call. Over the course of 28 days (epoch) all fees are collected in the WithdrawQueue contract and then once an epoch has passed, users can claim their fees proportional to their owned shares and the overall amount of shares.
Furthermore, this contract handles various request modifications such as cancellation and forfeiting which can only be triggered by the StakingVault directly. In-depth processing explanations can be found in the corresponding appendix below.

Appendix: pendingQueue processment

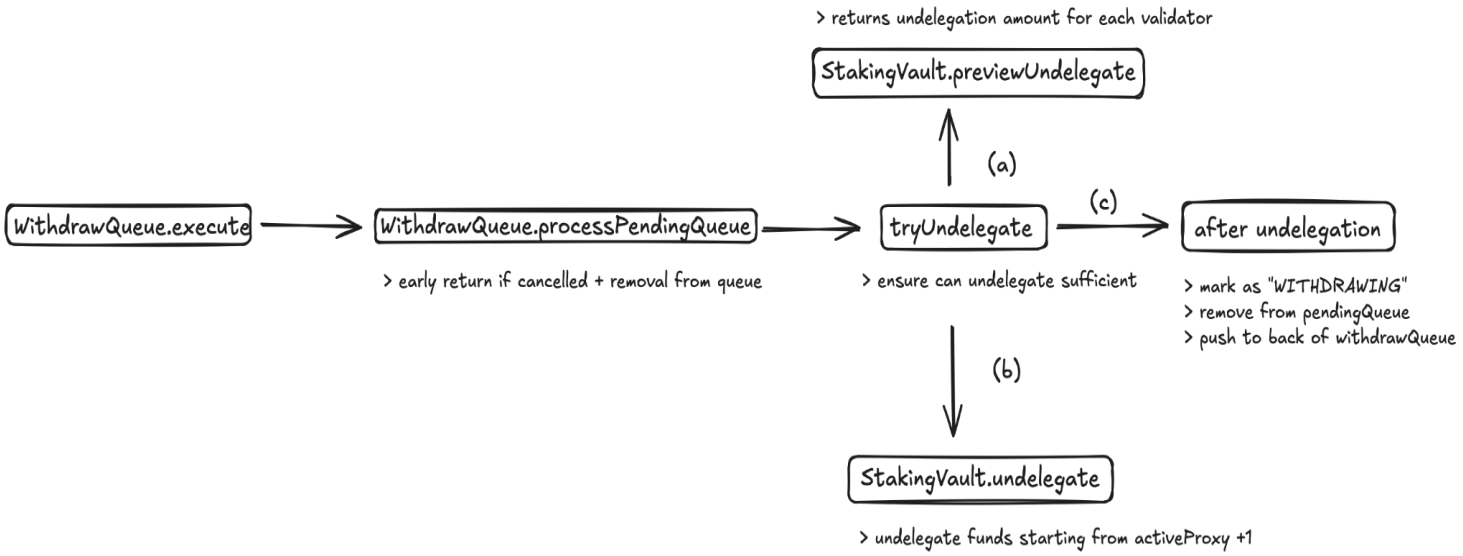
As mentioned above, the pendingQueue is the first queue which contains all initially requested redemptions via the enqueue function and works using the FiFo method. Requests in the pendingQueue will be processed as follows:

a) Via previewUndelegate it is fetched how much of each individual validator should be undelegated, which is based on the overall requested amount and the proportional share of each validator on the overall delegated amount, while only unlocked validators are counted. A full explanation of the previewUndelegate function can be found within the StakingVaultUpgradeable section

b) In the next step, the undelegate function in the StakingVaultUpgradeable contract is invoked which loops over all proxies and the corresponding validators while ensuring that the target undelegation amount for each validator is met. It is important to mention that this process starts from the first proxy after the current active proxy and loops over all proxies until the target amount for each validator is met. This has the result that the activeProxy +1 will experience a proportionally higher undelegation than the subsequent ones.

   During undelegation, funds are withdrawn from unlocked validators into the staking balance via the L1Write.sendTokenDelegate system call and then from the staking balance withdrawn to the spot balance via the L1Write.sendCWithdrawal system call.
   It is important to mention that funds only arrive after 7 days in the spot balance and will be accounted for via the totalPendingWithdrawal variable during that time.

This process can be illustrated as follows:



Appendix: withdrawQueue processment

Once a request within the pendingQueue has been executed it will automatically be pushed into the withdrawQueue during the processPendingQueue function. At this point it is expected that all desired funds are in the process of undelegating or have already arrived on the L1 spot balance. Requests in the withdrawQueue will be processed as follows:

a) The received asset amount is calculated based on the current ER and the smaller of the current result and the result at the time where the request has been prepared.

b) The prepareRequestRedeem function on the StakingVault is invoked which returns how much funds are currently available on the EVM chain for redemption request. At this point it is possible that the undelegation has not been fully executed and funds are still in the 7 day transition period. Under any circumstance, the withdrawSpot system call is executed which will transfer all spot balances from all proxies to the EVM chain. It is important to mention that under all circumstances, the prepareRequestRedeem function is called multiple times for a redemption. The following scenarios are possible:

1) prepareRequestRedeem is called while the undelegation has not been processed - nothing happens.
2) prepareRequestRedeem is called while the undelegation has been processed and funds are sitting in the L1 spot - these are withdrawn and available to be fully finalized upon the next execution
3) prepareRequestRedeem is called while the undelegation has been processed and the funds have been transferred from L1 spot to EVM proxy balances - the request can now be fully finalized
4) prepareRequestRedeem is called while the undelegation has not happened but the EVM proxies have received sufficient funds due to a forfeited request

c) The prepareRequestRedeem function is now called again as described above (after funds have now arrived on the EVM chain). The return value of this function is now sufficient to honor the desired requested assets.

d) The finalizeRequestRedeem function is now called because the requested assets can be fulfilled which calculates the redemptionFee and invokes the finalizeRequestRedeem function on the StakingVault.

e) The finalizeRequestRedeem function on the StakingVault now handles the storage updates which allows the controller to honor his request as well as transfers fees to the WithdrawQueue contract and burns the corresponding shares amount. The increase of the totalClaimableAssets variable is a very important step as this ensures that these withdrawn assets remain ready for the user to withdraw under all circumstances.

Core Invariants:

INV 1: The enqueue, cancel, forfeit functions must only be callable by the StakingVault contract

INV 2: A new request must have the PENDING status

INV 3: A request can only be cancelled if it has the PENDING status

INV 4: A request can only be forfeited if it has the WITHDRAWING status

INV 5: Fees can only be claimed for past epochs

INV 6: Each epoch is 28 days

INV 7: The execute function can only be called once per block

INV 8: Fees are only accrued if one of both or both queues are occupied

INV 9: The processPendingQueue function must return early if a requestId has been cancelled

INV 10: The processPendingQueue function must not pass if undelegation for the desired amount is not possible

INV 11: A requestId is marked as WITHDRAWING after it has been removed from the pendingQueue and pushed into the withdrawQueue

INV 12: The processWithdrawQueue function must only pass with requestIds which are marked as WITHDRAWING or FORFEITED

INV 13: A requestId is only finalized if the StakingVault has sufficient assets to honor the request

INV 14: All requestIds must be increasing and unique

INV 15: The enqueue function must revert for zero asset amounts

INV 16: The execute function must process both queues if non-empty

INV 17: Forfeited requests must still be withdrawn to the StakingVault

Privileged Functions

- setRedemptionFee

Issue_33	Lack of edge-case support for FORFEITED request allows for draining fees from the contract
Severity	**High**
Description	Within the finalizeRequestRedeem function, the fee is determined as follows and flows into the totalFees accounting for each epoch: *uint256 fee = Math.min((amount * $.redemptionFee.fee) / 10000, $.redemptionFee.maxAmount);* *$.feeEpochs[feeEpoch()].totalFees += fee;* The problem is that for FORFEITED requests, users will actually get their full shares back (if fixed) and no fee is being sent to the WithdrawQueue contract: *if (forfeited) {* *// if the request was forfeited then we just finalize the withdraw* *// and allow the assets to be redeposited at a later stage* *require(finalizeWithdraw(assets) == 0, FinalizedAmountNotEnough());* However, the totalFees variable was already increased. This is not only a logical issue in itself with multiple side-effects such that some users won't be able to claim their fees but can also be exploited in the scenario where the forfeiting logic is fixed, as users can repetitively request and forfeit just to inflate the fees

	and then claim them before other users claim them and thus exploit the inflated value and leaving other users left with nothing.
Recommendations	Consider simply accounting for forfeited requests and do not increase the totalFees variable in that scenario.
Comments / Resolution	Resolved, the whole forfeiting logic has been removed. It has to be noted that the fee mechanism has been refactored.

Issue_34	Fee change will impact already queued positions
Severity	**Medium**
Description	Fees can be changed via the setRedemptionFee function. These new fees will impact already enqueued positions. If users are not aware of the fee change they may not be able to cancel their request and thus experience a higher than expected fee.
Recommendations	Consider communicating the fee change.
Comments / Resolution	Acknowledged.

Issue_35	Lack of fee validation
Severity	**Medium**
Description	The setRedemptionFee function allows for setting the fee. Currently, there is no validation which ensures that the setting will result in a reasonable value.
Recommendations	Consider adding reasonable validation.
Comments / Resolution	Resolved.

Issue_36	Pausable logic not implemented
Severity	Medium
Description	The contract inherits the PausableUpgradeable modifier but does not actually use it. This means the WithdrawQueue contract has no emergency pausing features.
Recommendations	Consider implementing the whenNotPaused modifier.
Comments / Resolution	Resolved.

Issue_37	Undelegation happens with truncated decimals
Severity	Low
Description	The undelegate flow is always executed in wei (8 decimals). However, the finalizeRequestRedeem call is done with 18 decimals: *uint256 amount = Math.min(entry.assets, $.stakingVault.convertToAssets(entry.shares));* In the scenario where the StakingVault has no dust left (if for example dust was 999999999 and now became 10000000000 the dust is compounded), it can happen that users cannot claim their request or claim some funds from other users. This issue has only been rated as low severity since users are not enforced to redeem all their shares and can simply leave a few wei left to still honor their withdrawal.
Recommendations	Consider zero'ing out the last 10 decimals for the finalizeRequestRedeem function. Moreover, deposits should not be allowed with full precision in order to avoid dust in the first instance already.

Comments / Resolution	The withdrawal/undelegation flow has been fully refactored. This must be fully audited and formally verified with all possible edge-cases in order to mark as resolved in an effort to ensure no side-effects are introduced due to the new implementation. This status is subject to update.

Issue_38	Unclaimable fees in case processor has no fallback function
Severity	**Low**
Description	The claimFees function force transfers native ETH via a simple call: *payable(account).sendValue(totalFees);* This may not work if the processor has no fallback function.
Recommendations	Consider simply wrapping it to WNative if the call reverts.
Comments / Resolution	Acknowledged.

Issue_39	Incorrect order of operations prevents requestId from being properly removed
Severity	**Low**
Description	Currently, the processWithdrawQueue function is deleting the storage entry: *delete $.entries[requestId]* before using entry.controller to remove the request from $.requests[entry.controller]. Once the delete happens, the entry struct fields (including controller) become zeroed out, so entry.controller is no longer reliable and thus the removal is not executed.
Recommendations	Consider removing the requestId before the storage is adjusted.
Comments / Resolution	Resolved.

Issue_40	Unused error
Severity	**Informational**
Description	The contract contains one or more unused errors which will confuse third-party reviewers and increases the contract size for no reason: *error InsufficentAmountForUndelegation(uint256 expected, uint256 actual);*
Recommendations	Consider using or removing this error.
Comments / Resolution	Acknowledged.

Issue_41	Fee is not set upon initialization
Severity	**Informational**
Description	Currently, the redemption fee is not set during the contract initialization. Users can thus initiate requests without paying any fee and even worse if users do that while the fee is set during the redemption process, they will have to pay unexpected fees.
Recommendations	Consider initializing the contract with a fee.
Comments / Resolution	Acknowledged.

Issue_42	Redundant condition check within tryUndelegate
Severity	**Informational**
Description	The tryUndelegate function has the following condition check: *if (total >= WeiMath.toWei(assets))* This is redundant due to the fact how the previewUndelegate function computes the undelegations. Even if there are insufficient funds to cover the request, it will still always return the _wei amount due to the percentage-wise calculation based on _wei as base. There might be scenarios where this is not true due to some truncation in the percentOf logic but this is part of another bug.
Recommendations	Consider revisiting this part of the codebase, increasing test-coverage and reconsider the necessity of this condition check.
Comments / Resolution	Acknowledged. However, as mentioned, the withdraw/undelegate flow has been refactored and this must be fully audited and formally verified with all possible edge-cases in order to mark as resolved in an effort to ensure no side-effects are introduced due to the new implementation.

Issue_43	PausableUpgradeable contract not initialized
Severity	Informational
Description	The PausableUpgradeable contract is inherited but not initialized. While this does not expose any harm, it is considered as best practice to initialize all inherited contracts.
Recommendations	Consider initializing the PausableUpgradeable contract.
Comments / Resolution	Resolved.

Validator

ValidatorSetLib

The ValidatorSetLib contract is a simple helper library which is used by the StakingVaultUpgradeable contract in an effort to manage the validator settings. It exposes functionality to find, add and update validator setups. A typical validator configuration consists of the enabled boolean and the targetWeighting parameter.

Core Invariants:

INV 1: A validator can only be added once

INV 2: During addition, totalTargetWeighting must always be increased by targetWeighting

INV 3: During update, totalTargetWeighting must always be adjusted by new and old targetWeighting

Privileged Functions

- none

Issue_44	Disabled validators can have non-zero targetWeight
Severity	**Low**
Description	It is currently possible to disable validators via the update function but at the same time there is no check which ensures that the targetWeight for such a validator is zero. While this does not result in less delegated funds due to the dust delegation at the end of the loop, it will falsify the concept of targetWeights as the last validator will get significantly more funds than the other validators.
Recommendations	Consider ensuring that targetWeight is zero if a validator is disabled.
Comments / Resolution	Resolved, this has been refactored to only use the targetWeight to determine whether a validator is active. In case of a zero targetWeight, validator must not be considered as active.

Issue_45	Unused function
Severity	**Informational**
Description	The contract contains one or more functions which are completely unused. This will increase contract size for no reason and can confuse third-party reviewers. - tryFind - update [IMPORTANT] - targetWeighting
Recommendations	Consider elaborating the use of this function.
Comments / Resolution	Ackcknowledged.

Math

WeiMath

The WeiMath contract is a simple helper library which is used by the StakingVaultUpgradeable, ProxyLib and WithdrawQueueUpgradeable contracts.

It handles the conversion from 8 decimals (wei) to 18 decimals and vice versa as well as a percentOf function which calculates the percentage amount of a Wei value on the total Wei value:

> Wei * 1e18 / totalWei

And a mulPercent function which calculates a percentage share on a Wei value:

> Wei * percentageIn1e18 / 1e18

Privileged Functions

- none

Issue_46	Truncation during toWei
Severity	**Low**
Description	The toWei function truncates the last 10 digits from 18 decimals to 8 decimals, always resulting in a nominal value decrease. This can have unexpected side-effects on the overall codebase.
Recommendations	Consider carefully reviewing all states where toWei is used in context of potential truncation issues.
Comments / Resolution	Acknowledged.

Issue_47	Incorrect NATSPEC
Severity	**Informational**
Description	The documentation of this contract repetitively mentions that the USD amount is denominated in 6 decimals as well as mentions perps: */// @dev converts a wei amounts which is 8 decimals to a USD amount used in perps and vaults which is 6 decimals.* However, it is denominated in 18 decimals in all places on the L1 chain.
Recommendations	Consider updating the documentation.
Comments / Resolution	Resolved.

Lens

StakingVaultLensUpgradeable

The following changes have been identified as logical changes/OOS changes/intrusive changes that warrant an additional audit round; however, they may not account for all such changes introduced:

- **new getVaultQuery function**
- **refactoring of getProxyAccount function**

The StakingVaultLensUpgradeable contract is a simple lens contract which is compatible with multiple StakingVaultUpgradeable contracts using the same codebase. It allows for the following lens features:

a) Fetching of ER
b) Fetching of assets for all proxies
c) Fetching of assets for one proxy

Privileged Functions

- none

No issues found

Proxy

ProxyLib

The ProxyLib contract is a simple helper library which is used by the StakingVaultUpgradeable to fetch delegations, withdraw from the L1 spot balance and delegate/undelegate funds.

Appendix: Delegation Mechanism

The delegate function is called by the StakingVault whenever a deposit happens, ensuring that the corresponding amount is properly delegated to the desired validator for a proxy address.
At the point where this function is called, the Proxy on the EVM chain is already the owner of the funds. Afterwards, the following steps are executed:

a) Funds are bridged to the L1
b) Funds are moved from the spot balance to the undelegated balance
c) Funds are then delegated from the undelegated balance to a validator

These changes will be reflected in the next block.

Appendix: Undelegation Mechanism

The undelegate function is called by the StakingVault whenever an entry from the pendingQueue is processed. It will attempt to withdraw a very specific amount from the corresponding validator for the corresponding proxy. The following steps are executed:

a) Funds are undelegated from the validator to the undelegated balance of the proxy on the L1 chain

b) Funds are moved from the undelegated balance to the spot balance of the proxy on the L1 chain. This process undergoes a 7-day transition period where funds are sitting in the totalPendingWithdrawal state

In the next block, the undelegated balance will be reflected by the totalPendingWithdrawal balance.

Appendix: Delegation Fetching Mechanism

a) delegated: Delegated from the **staking balance** to a validator (after delegation - 24 hour waiting period)

b) undelegated: HYPE is sitting on the L1 chain in the **staking balance**, next steps are either withdrawing to spot balance (*withdrawing*) or delegating to a validator

c) withdrawing: HYPE is in the withdrawing process from the **staking balance** to the **spot balance**, locked for 7 days in this state and afterwards automatically granted to the **spot balance**

d) spot: HYPE is in the **spot balance** on the L1 chain and can either be moved to the staking balance or bridged to the EVM chain

Core Invariants:

INV 1: All balances from the L1 chain must be returned with 8 decimals

INV 2: readDelegations must return the Delegation for all validators, sequential

INV 3: System interactions must not be executed with zero values

INV 4: During readDelegatorSummary, undelegated value must be always zero

Privileged Functions

- none

Issue_48	Zero value delegation may result in unexpected side-effects or reverts
Severity	**Medium**
Description	Within the withdrawSpot and undelegate functions, a zero check is present to prevent unexpected side-effects from invoking system functions with a zero amount: *if (_wei > 0) {* *proxy.execute(address(SYSTEM_WRITE), abi.encodeCall(L1Write.sendSpot, (SYSTEM_ADDRESS, token, _wei)));* *}* Such a check is missing during undelegate due to the assumption that the provided value will never be zero. This however does not always hold true, as it is possible that a validator has no targetWeight and thus the delegation will be zero. In that scenario, a zero delegation would be attempted which can result in unexpected side-effects or even worse in reverts. This is amplified due to the fact that validator settings are currently not changeable.
Recommendations	Consider adding a zero value check before execution.
Comments / Resolution	Resolved, such a check has been implemented.