BAIL security

Hyperdrive
LST

FINAL REPORT

August ‘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.

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	https://github.com/ambitfi/hyperdrive-contracts/tree/634993b472c73b3ef613c72b0f273d2ce1f73b92/packages/staking/contracts/protocol
Resolution 1	https://github.com/ambitfi/hyperdrive-contracts/tree/498892de0a1199bf645207ab408b321521193fc5

2. Detection Overview

Severity	Found	Resolved	Partially Resolved	Acknowledged (no change made)	Failed resolution
High	3	3			
Medium	6	3		2	1
Low	10	8		2	
Informational	5	4		1	
Governance	1			1	
Total	25	18		6	1

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 post 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

3. Detection

StakingVaultUpgradeable

The StakingVault contract is the entry point that allows for staking HYPE and delegating to validators on Hyperliquid.

The contract implements ERC7535 to handle native assets and ERC7540 to handle asynchronous ERC4626 operations, a vital feature to handle the redemption process to undelegate, unstake, and withdraw HYPE from Hypercore.

Users deposit HYPE on the StakingVault, which forwards the deposits to the CoreController, who is in charge of distributing the HYPE to intermediate contracts called Proxies, which are used for the actual staking and delegation to different validators by distributing the delegation amounts based on the corresponding weight that has been defined to each of them.

StakingVault is in charge of handling the redemption lifecycle of staked HYPE, from the undelegation, unstaking, to the actual withdrawal from the L1 until finally the native HYPE can be delivered to the depositors.

Appendix: Deposits

When new deposits are made, all the free HYPE balance is staked on the active proxy determined by the CoreController, which is used to delegate to a set of validators by delegating based on the assigned weight to each validator.

Appendix: Withdrawals

The vault explicitly forbids withdrawals by specifying an amount of assets because of the nature of the share-based redemption; instead, withdrawal of HYPE is only allowed via the redemption of shares.

Appendix: Redemptions

The redemption of shares is an asynchronous process that involves undelegation, unstaking, withdrawal from Hypercore, finalization of batch requests, and actual claiming of assets for redeemed shares.

Appendix: Lifecycle of a request redemption

High-level overview of the redemption process:

- requestRedeem ⇒ undelegate ⇒ unstake ⇒ finalize ⇒ redeem

Multiple users' requests to redeem shares are packed together in the same batchRedemption

- Each user can request their redeemRequest to be queued on a specific controller's queue
- A requestId is claimable only after its status has been set to STATUS_FINALIZED , which indicates that the undelegation and unstaking have been completed, and assets have been withdrawn from the CoreController.

Appendix: Request Redemption Process

The lifecycle of a redemption starts at the requestRedeem().

Users request to redeem an amount of shares, which are tracked either on the current batchRedemption or a new one, depending on whether the current one has expired or not.

If there is no entry already registered for the requestId on the userRedemptions.requests queue, then this function queues the requestId into it.

Tracks the amount of redeemed shares for the requestId on the userRedemptions.shares, as well as in the batchRedemptions.shares

Appendix: Undelegation Process

The undelegation process occurs via execution of the execute function. When no batchRedemption is ready to be finalized, if the first requestId on the batchRedemptionsPendingQueue has expired, and its status is not UNSTAKING, then that batchRedemption can be undelegated.

The undelegation process happens in the tryUndelegate function. First, it determines how many assets are pending to be undelegated, and in case a partial undelegation has previously occurred, it previews how much can be undelegated from the CoreController. If something can be undelegated, it proceeds to undelegate the funds by calling the proxies via the CoreController to send a rawAction to undelegate.

- The total amount of undelegated assets is updated, and if all the assets of the batchRedemption have been undelegated, then the batch status is marked as UNSTAKING, otherwise, it is marked as UNDELEGATING.

Appendix: Unstaking Process

The unstaking process occurs via execution of the execute function. When no batchRedemption is ready to be finalized, and the status of the first requestId on the batchRedemptionsPendingQueue is UNSTAKING, that means that batchRedemption can be unstaked.

The unstaking process happens in the tryUnstake function. First, it validates that the CoreController can handle unstaking the entire amount of assets of the batch, and if so, proceeds to unstake them.

- The timestamp at which the unstaking happened is saved, and the status of the requestId is marked as FINALIZING.

When the unstaking occurs, the requestId is dequeued from the batchRedemptionsPendingQueue and is queued to the batchRedemptionsFinalizingQueue

Appendix: Finalization Process

The finalization process occurs via execution of the execute function. When the first requestId found on batchRedemptionsFinalizingQueue is marked as FINALIZING and the unstakingDuration has passed, that means that batchRedemption can be finalized.

Assets are withdrawn from the CoreController, and totalClaimableAssets reflect the amount of withdrawn assets (minus fees). Shares requested to be redeemed on this requestId get burnt, and the status of the requestId is marked as FINALIZED.

Appendix: redeem()

The lifecycle of redemptions ends with the redeem function. The controller's owner or operator can finally redeem the assets for the requestId found at the top of the userRedemption.requests queue.

Assets worth the redeemed shares are transferred to the specified receiver, and the total amount of redeemed shares, as well as assets, are deducted from the userRedemption.shares && userRedemption.assets

- If all shares of a requestId have been redeemed, that requestId is dequeued from the userRedemption.requests queue
- When all shares of the batchRedemption have been claimed, the data of that batchRedemption is deleted

Appendix: execute()

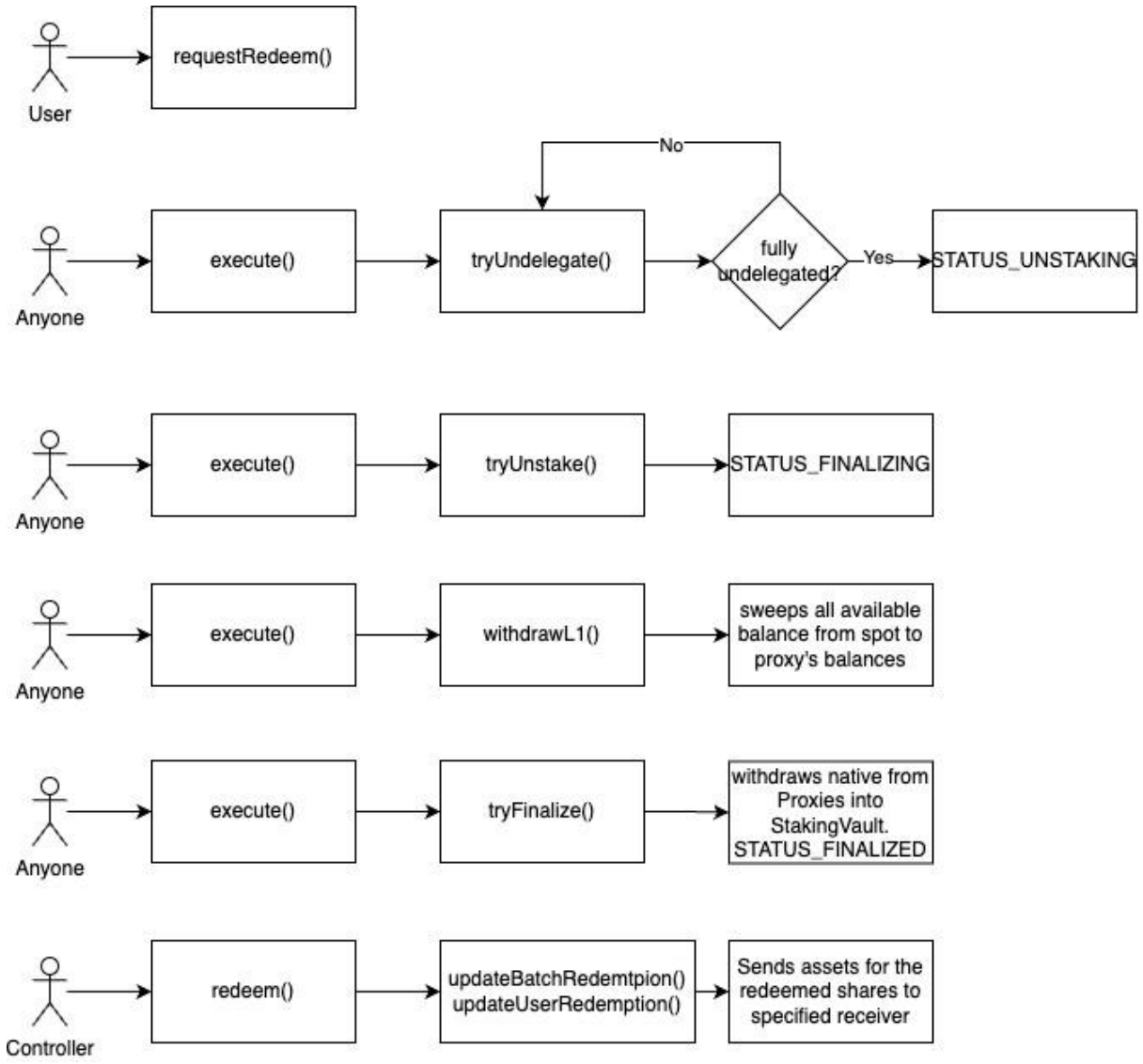
The execute() is a core entry point to handle and update the state of the requestIds given a set of conditions that determine whether the batchRedemptions can be finalized, unstaked, undelegated, or if there is anything to withdraw from the L1 via the CoreController.

- The action with the highest priority is to **unstake and finalize the first requestId found on the finalizingQueue.**
    - This is the last step before allowing the redeemers to finally claim their assets. Native assets are pulled from the proxy's balances into the StakingVault.

- The second most important action is to **withdraw any balance sitting in the spot (withdrawL1())**
    - Withdrawing from the L1 into the Proxy's balances is a prerequisite before finalizing a batchRedemption because, thanks to the withdrawal from the L1, the Proxies will receive the required native balance to process the finalization of the batchRedemptions.

- The 3rd most important action is to **unstake** from the CoreController. This occurs only when all the assets of the batch requests have been undelegated
    - If requestID is not yet ready to be unstaked, execution ends because the remaining actions are meant for the unstaking part of the process.
    - If requestId is ready to be unstaked, then the StakingVaults requests the unstaking of the corresponding assets to the CoreController, who determines whether there is enough undelegations available to fulfill the unstaking request, and if so, proceeds to send an unstaking rawAction request from the Proxies where there are undelegated assets.

- The last action is to **undelegate** amounts from the CoreController
    - Undelegations are capped by the lock timestamp imposed at the Hypercore level. As time passes, the delegated assets are unlocked for undelegation.
    - Once all the assets of the batch have been undelegated, the batch is marked as STATE_UNSTAKING, which allows unstaking from the CoreController.
    - If only a partial undelegation of the total assets is undelegated, then the batch status remains as STATUS_UNDELEGATING, which means that requestId still has pending undelegations.



Appendix: Queue Mechanism

The StakingVault implements a mechanism to handle the state of the batchRedemptions by tracking and updating two main DoubleEndedQueues [batchRedemptionsPendingQueue && batchRedemptionsFinalizingQueue]

There is a third queue [userRedemption[controller].requests] that is used to track the requestIds made by users based on the specified controller that will be allowed to complete the redemption process.

The overall status of the requestId is tracked on the batchRedemptions mapping.

- batchRedemptions is in charge of tracking the accounting, as well as the current status of the requestIds

Appendix: Checks in the redemption process

When depositing funds, the contract stakes all deposits and immediately delegates them. During redemptions, the system is highly dependent on the return values of the previewUnstake and previewUndelegate functions. As long as these functions don't return the exact amount requested, the status of the redemption request does not change.

While this is a design decision, it leads to a number of issues as listed in this report. If any amount of funds is left undelegated or unstaked, withdrawals can stop, since there might not be enough funds to undelegate or unstake even if the funds are present in the contract. For example, the tryUndelegate function expects the entire specified amount to be undelegated, even if there are undelegated funds already present in the proxy. Similarly, tryUnstake expects the entire amount to be unstaked even if unstaked funds are already available in the proxy. This can create situations where dust amounts left undelegated in the contract can create issues during redemptions.

An alternative approach would be to put in looser restrictions on the system and utilize freed funds. Undelegations should take into consideration already undelegated dust funds, and unstaking should similarly consider already unstaked dust funds. This will increase the efficiency of the system and prevent a number of DOS issues where admins need to intervene to keep the system functioning properly.

Core Invariants:

- Not allowed to withdraw an amount of assets, only redeem an amount of shares
- Each deposit or mint triggers a stake action on the CoreController to stake the newly deposited native in the StakingVault
- totalClaimableAssets are not considered part of the totalAssets because they represent assets that are reserved for redeemers

Appendix: Invariants for the request redemption process

- All requests to redeem shares that are made on the same batchRedemption will be packed together for undelegation and unstaking from the CoreController.
- New requests while the batchRedemption of the lastRequestId is still valid will be added to the last created batchRedemption.
- When a new request comes in and the batchRedemption of the lastRequestId has expired, a new batchRedemption has to be created and initialized.
- Request redemptions for the same controller are packed together by accumulating all the shares that have been redeemed using the same controller.

- If the specified controller has 0 shares for the lastRequestId, the lastRequestId must be enqueued into the userRedemptions[controller].requests.

Appendix: Invariants for the undelegation process

- A requestId can be undelegated when no batchRedemption is ready to be finalized, the first requestId on the batchRedemptionsPendingQueue has expired, and the status of that batchRedemption is not UNSTAKING.
- If all the assets of the batchRedemption have been undelegated, then the batch status is marked as UNSTAKING; otherwise, it is marked as UNDELEGATING.

Appendix: Invariants for the staking process

- A requestId can be unstaked when no batchRedemption is ready to be finalized, and the status of the first requestId on the batchRedemptionsPendingQueueis as UNSTAKING.
- When all assets of a batchRedemption are unstaked from the CoreController, the requestId status is marked as FINALIZING.
- When the unstaking occurs, the requestId is dequeued from the batchRedemptionsPendingQueue and is queued on the batchRedemptionsFinalizingQueue.

Appendix: Invariants for the finalization process

- A requestId can't be dequeued from the finalizingQueue until the batchRedemption assets associated with the requestId have been fully undelegated.
- A requestId can't be finalized unless the unstakeDuration period has passed.
- requested shares to be redeemed during the batchRedemption is burned.
- requestId status is marked as FINALIZED.
- Assets are withdrawn from the CoreController and transferred into the StakingVault.
- totalClaimableAssets grows by the amount of withdrawn assets from the CoreController, minus fees.

Appendix: Invariants for redeem()

- The requestId found at the top of the userRedemptions.requests queue must have its status as STATUS_FINALIZED.
- Can redeem at most the amount of shares saved on the userRedemption.shares.
- Discounts from the totalClaimableAssets the amount of assets that will be sent to the receiver.
- Dequeues the requestId from the userRedemptions.requests queue if all assets of the controller for that requestId have been redeemed.

- If all the assets of the batchRedemption for the requestId have been redeemed, then delete the data of the batchRedemption for that requestId.

Appendix: Invariants for Queues

- When enqueuing a new requestId to a DoubleEndedQueue, the item is added to the back.
- When dequeueing from a DoubleEndedQueue, the item at the front is popped.

Appendix: Invariants for batchRedemptionsQueue

- requestIds are queued in this queue whenever the value of lastRequestId increments.
    - That is when the batchRedemption for the current lastRequestId has expired
- requestIds are dequeued from this queue when all the assets of the batchRedemption of the tokenId have been fully unstaked from the CoreController.

Appendix: Invariants for batchRedemptionsFinalizationQueue

- requestIds are queued in this queue after the requestId has been unstaked
- requestIds are dequeued from this queue when the requestId has been finalized

Appendix: Invariants for users.requests queues.

- A new requestId is queued in this queue when requesting a new redemption, and there are 0 shares for the lastRequestId
- A requestId is dequeued from this queue when all the shares of the controller for that requestId have been redeemed

Privileged Functions

- setMaximumSupply
- setRedemptionFee
- setMinimumRedemptionAmount
- setBatchDuration
- setUnstakingDuration
- pause
- resume

Issue_01	A drop in the exchange rate for a requestId that has been partially undelegated can DoS the entire redemption system for all the existing requestIds
Severity	High
Description	requestIds can partially undelegate HYPE from the Validators. batch.assets is computed based on the exchange rate per assets/shares. The first time a requestId is undelegated, batchAssets is computed based on the exchange rate at that moment. If all the assets are not undelegated, that requestId status will remain as UNDELEGATING and a second undelegation has to occur in order to fully undelegate all the assets. The issue is that when the exchange rate drops while there is an active partial undelegation for a requestId the system can fall into a DoS that would break the redemption mechanism for all the requestIds. Let's follow a requestId for: batch.shares == 1k. - During the first undelegation, suppose the exchange rate is 1:1, and there are 900 assets available for undelegation. This will leave the system after the first undelegation as: ○ batch.assets == 1k ○ batch.undelegated == 900 - Now, before the next undelegation, the exchange rate drops by 20%. This will recompute batch.assets and set to be the minimum between the current value of batch.assets, which is 1k, and the new value for the batch.shares, which is 800. - Therefore, batch.assets will be updated to 800. When the remaining amount is computed in tryUndelegate, the result is WeiMath.ZERO because batch.undelegated > batch.assets. As a result, coreController.previewUndelegate(remaining) returns 0, which leads to an early false return. *if (total == WeiMath.ZERO) {*

	return false; *}* Thus, this requestId will be forever stuck in the undelegation phase and even block requests after it, blocking the entire queue.
Recommendations	As part of the validation to determine if execution has to return early from tryUndelegate(), also verify if remaining > WeiMath.ZERO
Comments / Resolution	Fixed - Returns early from tryUndelegate() if undelegated >= batch.assets

Issue_02	Multiple execute transactions in the same block can break accounting
Severity	High
Description	The execute function does unstaking/undelegating operations on queued withdrawals. This function can be called multiple times in the same block, and reads data from the hypercore in the previewUndelegate and similar functions. The issue is that if 2 execute transactions are in the same block, it will read the same hypercore state, without registering any changes due to the transaction itself. As an example, consider a situation where a batch is trying to undelegate 100 tokens, but there's only 60 available for undelegation; the rest are on cooldown. So, previewUndelegate(100) returns 60 as total, which is stored. Now, if another execute transaction is run in the same block, the hypercore state is still the same. In the 2nd transaction, previewUndelegate(40) will be run (since 60 was already recorded as undelegated), and since 60 tokens are still available in the hypercore, this call will return the full 40 as the total. Now the system will mark this batch as undelegated and change the state to STATUS_UNSTAKING. However, on the hypercore side, only 60 tokens will get undelegated, and the 2nd undelegation of 40 tokens will revert since the rest are on cooldown. This breaks future withdrawals, since the program will now try to unstake and payout 100 tokens, but only 60 have been undelegated and available for unstaking/paying out. An admin has to intervene with a forceUndelegate to fix the accounting in the system. Since this is possible in every redemption batch, which happens every 12 hours, this can lead to constant DOS of the redemption system and constant manual intervention to clear it. Malicious users can even trigger this intentionally, constantly tripping up the system.
Recommendations	Consider allowing the execute function to only run once per block.

	Record the block.timestamp or block number and revert if this matches with the older value. The core issue is that this function makes changes to the hypercore, but cannot track its state since the state only changes after the block. So another way to mitigate this would be to record the block timestamp/number as well as the amount undelegated/unstaked in that block, and then deduct that from operations if in the same block. Similar to the approach used in the totalAssets function here, but it will be more complicated since this has to be repeated for undelegations as well as unstaking.
Comments / Resolution	Fixed - Added a modifier to track the block.number of the last time the function was called, and, enforce the current block.number to be > than the lastBlockNumber

Issue_03	Vault is susceptible to share inflation
Severity	**Medium**
Description	That vault mints shares to the user based on their deposit amounts and the totalAssets. The totalAssets is a function of the funds in the vault as well as funds elsewhere in the system. The issue with this design is that it is easy for any user to inflate the share ratio of the vault, and it is most vulnerable during times when the vault is nearly empty. Consider a situation when the vault is empty, so totalSupply and totalAssets are both 0. Now, a user deposits 1 wei of HYPE to the vault, so both totalAssets and totalSupply become 1. Now the user can make donations to the system to manipulate this 1:1 ratio. Some common ways to do this can be by self-destructing funded contracts in HyperEVM, transferring funds directly to the vault, or by raw sending HYPE tokens to the address of a proxy on Hypercore. Donating 1e8 HYPE tokens this way on Hypercore will keep the number of shares the same, but increase the totalAssets to 1e18+1 on the vault. When another user comes in and deposits less than 1e18 HYPE tokens, they get minted 0 shares, essentially losing their deposit to the initial depositor.
Recommendations	Consider locking the first 1000 shares minted from the vault on the first deposit. This will prevent share manipulation by massively increasing the cost of manipulation.
Comments / Resolution	Acknowledged - Initial shares will be minted and burnt.

Issue_04	No slippage checks during deposit/redeem operations
Severity	**Medium**
Description	The vault system is designed to take in funds from the user and mint out shares. However, the user can only specify one of the two parameters, assets or shares. The issue with this design is that since the internal share ratio is constantly changing, and can even be manipulated since it reads off raw balances of accounts, the user can be minted an unexpected number of shares or charged an unexpected number of assets. Say the user expects 1e18 shares for 1e18 HYPE tokens, so they call the deposit function with 1e18 as the amount of shares. However, their transaction was preceded by a rewards distribution, and so they only get 0.99e18 shares. This can be unexpected, and this difference can be large if the share ratios change by large amounts. Another example is the case in the previous issue, where users get minted 0 shares since there is no way to specify a minimum amount of expected shares to the system. This issue is valid for the deposit and requestRedeem functions only. The mint function is unaffected since users specify the amount of HYPE they send by controlling msg.value, and the withdraw function is turned off.
Recommendations	Common approaches to combat this are: 1. Creating new protected mint/requestRedeem functions on top of the existing functions, which also take in a parameter called minsharesOut or minAssetsOut. Since this is a separate function, this will not break ERC4626. 2. Pass user interactions through a separate router contract, like in Yearn vaults. The router functions handle slippage, keeping the vault contract still ERC4626 compliant.
Comments / Resolution	Acknowledged

Issue_05	Not accounting for msg.value on maxDeposit and maxMint causes the limit to be hit prematurely
Severity	**Medium**
Description	deposits and mints have a limit determined by the maximumSupply, which caps the total amount of native that can be supplied into the system. The issue is that the contract uses totalAssets, which already includes the funds sent to the contract. Say the max deposit is 100 and the current totalAsset is 99. So the contract should take in 1 HYPE more. However, when depositing 1 HYPE, in the deposit function, when maxDeposit is called, totalAssets is already 100 since it includes the sent msg.value. So it returns a maxDeposit value of 0 since the limit is hit, and the transaction reverts even though it should have passed.
Recommendations	Account for the msg.value on the deposit() and mint() - On deposit(), add msg.value to the output returned from maxDeposit(receiver) - On mint(), convertToShares(msg.value) and add it to the output from maxMint(receiver)
Comments / Resolution	Fixed - Discounting msg.value from totalAssets() on maxDeposit()

Issue_06	Position owners can cause losses to operators by front-running them
Severity	**Medium**
Description	The vault implements an operator model, where position owners can give permissions to other addresses to manipulate their positions on their behalf. In the requestRedeem function, there is a check for this operator role. *address from = isOperator(owner, msg.sender) ? owner : msg.sender;* If this operator role check fails, the funds are deducted from the msg.sender instead of the owner. This can be used by the owners to grief operators if the operators themselves are holding any funds. Say Alice has 100 shares and has designated Bob to be an operator who already holds 1000 shares. Alice can then ask Bob to send a requestRedeem transaction, and Bob will initiate a requestRedeem(100, Alice, Alice) transaction. Alice can front-run this transaction (by sending the tx before Bob does) and remove Bob as the operator. Now, the operator check will fail, and 100 shares will be deducted from Bob instead, queued up to be withdrawable by Alice. Operators will generally be bots checking on-chain for such transactions. This design makes it easy to abuse such bots by bundling transactions in a way that you can drain them.
Recommendations	Consider just reverting if the isOperator check fails. If the caller is not the operator of the owner, there is no point letting the transaction go through anyway.
Comments / Resolution	Fixed - Assets are pulled only from the owner, and access control was modified to enforce onlyOwnerOrOperator are allowed to requestRedeem on behalf of the specified owner.

Issue_07	Slashing can lead to incorrect state updates
Severity	**Low**
Description	The execute function carries out undelegate/unstake operations depending on the status of the batch. If the contract decides that the entire amount is available for undelegation via previewUndelegate, it switches the state to STATUS_UNSTAKING. The contract actually emits an event through the core writer contract. The emitted event then creates a transaction on hypercore, which does the actual undelegtion. The issue is that if a slashing event happens in between the event emission and the transaction inclusion in the hypercore block, this can cause false state updates. Due to the slashing, there might not be enough funds to undelegate, and so no funds will be undelegated, but the contract state will still be incorrectly changed to STATUS_UNSTAKING and will require admin intervention via a forceUndelegate. Hyperliquid chain has not yet implemented slashing, so it is not possible to fully confirm this issue at this stage. If slashing events are not included in the block, this issue will never arise however, if slashing transactions can ever slip between EVM block inclusions and their emitted event based transactions, there can be incorrect state updates. This will only be validated/invalidated once the slashing mechanism on the L1 is finalized.
Recommendations	Recommend acknowledging this issue since it might not be preventable and would always require admin intervention if slashing can slip between EVM block execution.
Comments / Resolution	Acknowledged

Issue_08	minimumRedemptionAmount can lead to stuck funds
Severity	**Low**
Description	The vault contract defines a minimumRedemptionAmount variable, which is the minimum amount of assets a user must redeem for the redemptions to go through. This means that if the user is left with less funds than this amount, they will never be able to get it out of the system. Say the minimumRedemptionAmount is 10 HYPE and a user deposits 100 HYPE. their stake grows to 108 HYPE and then they burn up some shares to take out 100 HYPE for other payments. Now their 8 HYPE is stuck in the system since its below the minimum, and cannot be withdrawn. The severity of this issue increases when the contracts are paused. Normally, the user would have been able to deposit 2 HYPE and then withdraw all the funds (8 remaining + 2 freshly deposited), but this path will be blocked if the deposits are paused in case of a migration. In such a scenario, the funds are not recoverable without admin help.
Recommendations	Consider allowing smaller value assets as long as they empty the user's account (remaining shares ==0).
Comments / Resolution	Acknowledged - The minimumAmount will be a small value to help prevent any abuse

Issue_09	Any user can trigger false event emissions
Severity	**Low**
Description	Events are used by indexers to track balances, portfolio positions, etc, and thus are generally required to be accurate to prevent off-chain systems from doing incorrect accounting. In the requestRedeem function in the contract, users can emit false events, which can throw off such off-chain systems. *emit RedeemRequest(controller, owner, requestId, msg.sender, shares);* The event RedeemRequest emitted contains the owner address. However, if the caller is not an operator of the owner, the funds will be deducted from the operator themselves and not the owner. However, the event emitted will still report the original passed in owner address as the owner. Thus, any user can spoof a withdrawal of another user by passing in the target's address in the owner field. The event emitted will be identical to a legitimate event emitted when the target withdraws their funds, leading to indexers based on events breaking.
Recommendations	Consider reverting if the isOperator check fails in the requestRedeem function. This will prevent false event emissions.
Comments / Resolution	Fixed - Implemented onlyOwnerOrOperator() modifier for access control

Issue_10	Broken pagination in getRedeemRequests causes incorrect results
Severity	**Low**
Description	Both overloads of the getRedeemRequests function ignore the start index during iteration, causing the returned entries to always begin from index 0, regardless of the start value. This breaks pagination logic and results in incorrect data being returned for any start > 0.
Recommendations	Use start + i as the index in the loop instead of just i to account for pagination.
Comments / Resolution	Fixed - Implemented recommended mitigation.

Issue_11	Slashing can lead to stuck funds
Severity	**Low**
Description	When funds are being redeemed, they go through undelegation and then unstaking processes. In each step, the amount of assets is calculated, and if it has dropped, the lower value is used. This can lead to funds being left undelegated but not unstaked. This can cause issues since redemptions expect every single wei of funds to be undelegated and unstaked, and do not consider any funds already undelegated/unstaked in the system. Say Alice starts a redemption process for 100 tokens. After undelegation one of the validators gets slashed, so she is now only allowed to unstake and claim 90 tokens. This leaves the system 10 tokens that were undelegated but not unstaked. Now Bob tries to remove all funds from the system. The system has 200 tokens remaining, but only 190 of them are delegated and 10 are undelegated. However, Bob's withdrawal will be stuck since previewUndelegate will always return 190 tokens since that is all that is available, and Bob will not be able to claim the remaining 10 tokens. An admin has to intervene with a forceDelegate to delegate back the 10 tokens so that Bob can undelegate 200 tokens in one

	go. So, every time there is a slashing while a batch withdrawal is being processed, there will be funds stuck in the various steps of the system, which need to be rescued by the admin.
Recommendations	Refer to the appendix section Checks in the redemption process. The system should consider the already undelegated and unstaked funds present in the proxies. Another option is to acknowledge this issue and just manually interfere as the admin on every slashing event.
Comments / Resolution	Fixed - tryUndelegate() now checks for all undelegated amounts of each Proxy.

Issue_12	redeem is missing whenNotPaused modifier
Severity	**Low**
Description	The redeem function does not include the whenNotPaused modifier. This oversight allows redemptions to proceed even when the contract is paused. In emergency or paused states (e.g., due to a critical bug, slashing event, or governance intervention), redemptions should be blocked to preserve protocol safety.
Recommendations	Add the whenNotPaused modifier to the redeem function.
Comments / Resolution	Fixed - Implemented recommended mitigation

Issue_13	pendingRedeemRequest doesn't follow the ERC-7540 standard
Severity	**Low**
Description	According to EIP7540, pendingRedeemRequest must return all shares pending redemptions, excluding any shares which are already available for claim. The contract here uses multiple states to mark pending requests but the pendingRedeemRequest currently only returns shares marked as STATUS_PENDING. UNDELEGATING, UNSTAKING and FINALIZING are also pending requests and thus these should also be included in this function according to the EIP.
Recommendations	Return 0 only when status is 0 or STATUS_FINALIZED, any other status of the requestId should return the shares.
Comments / Resolution	Fixed - Implemented recommended mitigation

Issue_14	Missing admin input checks
Severity	**Informational**
Description	It is standard practice to check the inputs provided by the admin, which form the parameters of the system. Values such as redemptionFee, batchDuration, etc, are generally capped with a hard-coded value in order to prevent admin mistakes from throwing off the system too much. The contracts here do not check if the admin-provided values are within any such ranges.
Recommendations	Consider hard-coding certain ranges and checking if the admin-provided system parameters fall in those ranges in the setter functions.
Comments / Resolution	Fixed - Implemented recommended mitigation

Issue_15	Unused operators mapping in StakingStorage
Severity	Informational
Description	The StakingStorage struct defines an operators mapping. However, this field is never read from or written to in the StakingVaultUpgradeable contract, and the operators logic is instead inherited from ERC7540OperatorUpgradeable, which stores a different storage variable in a different location for its operations.
Recommendations	Remove the operators mapping from StakingStorage
Comments / Resolution	Fixed - Operators mapping has been deprecated

Issue_16	Duplicate code to withdraw from L1
Severity	Informational
Description	The contract calls withdrawL1 twice in the execute function, once at the top and once at the bottom. However, the lower call is redundant since there are no state changes in between, which can lead to more HYPE in the contract within the same transaction. This will lead to unnecessary extra events and transactions on the L1.
Recommendations	Remove the second instance of withdrawal from L1.
Comments / Resolution	Fixed - Implemented recommended mitigation.

Issue_17	finalization of batch redemptions exposes redeemers to slashing without earning yield.
Severity	Informational
Description	When finalizing a batch, the amount of assets to be finalized is derived from the minimum between the snapshot batch.assets at the moment of the unstaking, or the current value of assets based on the number of shares. Due to this, the batch will not be accruing any more yield, while still fully slashable.
Recommendations	We recommend acknowledging this issue and documenting this behavior because this is an accepted outcome in staking protocols.
Comments / Resolution	Acknowledged - Expected behavior

Issue_18	Redundant check when redeeming all remaining shares of a requestId
Severity	Informational
Description	This check is redundant because when all shares of a batchRedemption are being redeemed, all the remaining batch.assets will be given to the last redeemer. In case of any dust funds remaining, this can actually revert a transaction.
Recommendations	We recommend removing this check since all batch.assets are given to the last redeemer because shares being redeemed will be exactly batch.shares.
Comments / Resolution	Fixed - Implemented recommended mitigation.

ERC7535Upgradeable

An implementation to allow managing native assets with ERC4626 tokenized vault capabilities.

Has implemented logic for the deposit and mint to allow deposits of native assets.

Has logic to handle the conversion between shares to assets, and vice versa, as well as implementations to get the max allowed values for deposits, mints, redemptions, and withdrawals.

Issue_19	Missing check in the mint function allows minting without depositing funds
Severity	**High**
Description	The mint function mints shares to the user based on the passed-in shares value. It calculates the corresponding assets as well, in case refunds are to be made. However, there is no check to make sure the required assets amount is passed in. This means any user can call mint with an arbitrary number of shares and pass in 0 as msg.value. This will still mint the shares to the user, but for free.
Recommendations	Add a check require(msg.value>=assets).
Comments / Resolution	Fixed - Implemented recommended mitigation.

DoubleEnqueuedUpgradeable

A library that allows the manipulation of DoubleEndedQueues.

Facilitates accessing the front of the queue, as well as enqueuing and dequeuing values in the queue.

Also exposes a function that allows the retrieval of values from the queue.

No issues found.

CoreControllerUpgradeable

The CoreControllerUpgradeable is in charge of managing the staking, delegation, unstaking, undelegation, withdrawal from L1, and withdrawal from Proxies to StakingVault for all the native staked via the StakingVault

The CoreController maintains the set of Proxies and Validators that are involved in the staking and delegations activities.

Appendix: Staking

The CoreController determines the Proxy that will receive the native assets to be unstaked. It picks a Proxy from the list of proxies. The chosen proxy receives the amount of native being unstaked and sends a rawAction to stake that balance.

Appendix: Delegating

The CoreController delegates the task of determining the amounts to be delegated to each validator to the DelegationStrategy contract, which distributes the amount to delegate to each validator based on the assigned weight to each validator. Once the amounts to delegate have been calculated, the CoreController sends a rawAction to delegate from the active Proxy at the moment.

Appendix: Undelegating

- The CoreController also delegates the task of determining the amounts to be undelegated to the DelegationStrategy. To calculate the amounts to be undelegated from each validator, it is required to read the delegation's data from all the validators among all the proxies and extract only the delegations that have surpassed the lockedUntilTimestamp threshold. Once the undelegation amounts to undelegate from each validator are ready, the CoreController proceeds to iterate over each proxy and each validator to request an undelegation for the computed amount. To do so, a rawAction to undelegate is sent from each Proxy for each Validator.

Appendix: Unstaking

- The CoreController determines the amount to unstake from each Proxy by reading the delegator's summary. It will attempt to withdraw the most available of undelegated assets from each proxy until it fulfills the requested amount to be unstaked.
    - For the unstaking to occur, assets must have been undelegated first.
- The CoreController will make a call from each proxy to send a rawAction to unstake.

Appendix: Withdrawal from L1

- Once assets have been unstaked, they will still be on the Hypercore(L1). A request to pull the native from the spot onto the Proxy's balances has to be made.
- The CoreController is in charge of making such a call to each Proxy whenever they have available balances on the L1; that balance is pulled onto the Proxy's balance to enable finalization of request redemptions.

Appendix: Withdrawal

- The CoreController is in charge of sweeping assets from the Proxy's balance onto the StakingVault to enable the finalization of request redemptions.

Appendix: Force actions

- There are 4 functions that allow the admins to step in to fix issues that arise from failed executions on the Hypercore. For example, if an undelegation reverts at the Hypercore level, the admins can step in to request the undelegation manually.

Privileged Functions

- forceStake
- forceUnstake
- forceDelegate
- forceUndelegate
- withdraw

Issue_20	Consider skipping 0 amount L1 calls
Severity	**Low**
Description	The CoreController contract emits events, which are then used by the hyperliquid nodes to include transactions in the hypercore. However, certain transactions don't need to be included since they have no impact. The system already skips core writer calls for a number of operations if the amount in question is 0. However, the stake function and the withdrawL1 function still do core writer calls even when the passed-in amount is 0 wei. Otherwise, this can lead to large amounts of calls from the EVM block to the L1, which might consume a large amount of the gas limit of the hypercore block for 0 value transactions.
Recommendations	Consider implementing an if (_wei > WeiMath.ZERO) check in the stake and withdrawL1 functions.
Comments / Resolution	Fixed - Implemented recommended mitigation

Issue_21	Admin can drain all funds from the system
Severity	**Governance**
Description	The admin can grant themselves the OPERATOR_ROLE and call unstake along with forceUndelegate to divert any funds in the system to any address.
Recommendations	Consider giving the operator role only to timelocked multisigs. This reduces the fallout in case of an admin private key leak.
Comments / Resolution	Acknowledged - Admin will be a multisig

DelegationStrategyUpgradeable

DelegationStrategy is in charge of calculating the delegations and undelegations amounts on behalf of the CoreController.

Appendix: previewDelegate

- previewDelegate distributes the amount of native to be delegated to validators based on the assigned weight to each validator.

Appendix: previewUndelegate

- previewUndelegate is in charge of calculating the amounts to undelegate from each validator on each Proxy of the CoreController.
- It first reads the delegations that each validator on each proxy has, then uses only the delegations that are ready to be unlocked, that is, beyond the lockedUntilTimestamp threshold, and finally, based on the available undelegations, it distributes the amount to be undelegated from all validators of all proxies based on their available undelegations and the total available undelegations among all validators and proxies.

Core Invariants:

INV 1: Delegated amounts are distributed among all the validators based on the weight each validator has assigned.
INV 2: Undelegated amounts are distributed evenly among all validators on all proxies based on the total available undelegations and the available undelegations on each validator for each proxy.

Privileged Functions

- setTargetWeighting

Issue_22	previewUndelegate does not exclude the current active proxy
Severity	Medium
Description	In hyperliquid staking, delegations have a lockup of 1 day. This means that if a user delegates their funds to a validator, they cannot undelegate them for at least 1 day, and the funds are frozen. In the hyperliquid chain, hyperEVM blocks are minted with hypercore blocks in between. Delegation information is submitted in a hyperEVM block, but they arent actually executed until the next hypercore block is minted. So if a hyperevm block has 2 delegation/undelegation transactions, the second transaction will not be aware of the first transaction in any way, since the delegation change has to actually be applied in the next block. This means if in a hyperevm block there is a delegate transaction followed by an undelegate transaction for the same address, the second undelegation transaction might fail due to the previously mentioned 1-day lockup. And there is no way to check the state of the chain to check for this failure since the delegations won't happen until the next block. This idea is also mentioned in the comment present in the contract here; however, it is not implemented. So, if user A causes a delegation in proxy A, and then user B tries to trigger an undelegation on the same proxy in the same block, user B's undelegation will fail. So even though previewUndelegate will return as if the transaction was fine, the actual undelegation will never happen, and the user's redemption flow will move on to the unstaking phase. But unstaking won't work since the funds are still delegated, and this will brick the user's redemption until an admin intervenes and calls forceUndelegate. There is a delegation lockup check in the previewUndelegate function, but this will not work since the delegation transaction will not update the delegation state until the next hypercore block. Thus, there is no way to check on-chain to prevent this scenario.
Recommendations	Skip the current active proxy when processing previewUndelegate. However, this will deny any funds present in that proxy, which can

	prevent redemptions. This can be a major problem if there is only one active proxy.
Comments / Resolution	Issue not fixed. Deposit and execute calls can still take placein the same block, which would revert undelegation attempts fron the current active proxy. Since undelegation undelegates from every proxy which isnt locked, this will always lead to the undelegation call for the active proxy to fail, if the block had a deposit transaction already. Would require multiple attempts to undelegate amounts, leading to slower/stuck redemptions.

Issue_23	Rounding in previewUndelegate can block redemptions
Severity	**Medium**
Description	The redemption process of the vault happens in the following way: 1. user order redemption of say 100 assets. 2. 100 assets are undelegated. Until the entire amount (100) is undelegated, the system stays in this state, repeating the undelegation attempt 3. 100 assets are unstaked. 4. 100 assets are available for withdrawal. The point to note is that if 100 assets are present in the system but only 99 of them are delegated, the redemption will be stuck looping in step 2. The admin needs to intervene and forceDelegate the 1 asset, and then step 2 will be cleared. Till then, the redemptions will be blocked. This situation is created by the previewUndelegate function itself. It calculates the amount of tokens that need to be undelegated based on the current delegated amounts. *Wei amount = amounts[i][j] == WeiMath.ZERO ? WeiMath.ZERO : _wei.mulPercent(amounts[i][j].percentOf(totalAmount), Math.Rounding.Ceil);* However, due to the Math.Ceil present above, it overestimates the undelegation amount by a few wei every time. Say there are 2 validators, each delegated 100 assets. So totalAssets = 200. Say _wei, the amount to be withdrawn is 50. The 25 will be undelegated from each validator. But say validator A has 99 delegated and validator B has 101 delegated. In this scenario, 25 will be undelegated from validator A and 26 from validator B. So in total 51 assets will be undelegated, while only 50 assets were required to be withdrawable. This leaves the system with a state where validator A and B have 149 assets

	delegated, and there's 1 undelegated asset in the contract. Now, if a user tries to withdraw the entire amount, the system will be stuck in step 2 since previewUndelegate(150) will never return 150 as the total, at max 149. So the redemptions will be blocked until an admin force delegates the undelegated 1 asset in the contract.
Recommendations	In the previewUndelegate function, consider tracking a remaining variable that gets decremented starting from _wei. Then, the amount to undelegate, amounts[i][j], can be incremented by min(remaining,amount). This will prevent any extra undelegation. Also, refer to the appendix section Checks in the redemption process.
Comments / Resolution	Fixed - Implemented recommended mitigation

Issue_24	Incorrect total amount calculation
Severity	**Low**
Description	During undelegation, each validator's delegated amount is queried, and an undelegation amount is calculated based on their current delegation share. To cap the delegation amount, a last-minute Math.min is run. *amounts[i][j] = WeiMath.min(amount, Wei.wrap(delegations[i][j].amount));* However, the total amount calculated ignores this value and uses the old amount instead. *total = total + amount;* Thus, if the Math.min statement ever results in a reduced amount, the total will be off since it does not take into consideration the capped value.
Recommendations	Change the update to total = total + amounts[i][j].
Comments / Resolution	Fixed - amount and amounts[i][j] are now the same

Issue_25	Division before multiplication leads to allocation inaccuracy in previewDelegate
Severity	**Low**
Description	In previewDelegate, the contract calculates the share of delegation for each validator based on their weight, but performs division before multiplication, which causes early truncation and precision loss: *uint256 percentage = (weightings[i] * 1e10) / totalWeighting;* *amounts[i] = WeiMath.toWei((Wei.unwrap(_wei) * percentage));* This approach truncates the percentage to an integer before applying it to the total amount (_wei). The result is that each allocation is slightly underweighted, with a cumulative remainder that is only corrected at the end by adding it all to the last non-zero validator. This leads to inaccurate weighted delegations, as this is used when delegating to different validators.
Recommendations	Refactor to: *amounts[i] = WeiMath.toWei((Wei.unwrap(_wei) * weightings[i] * 1e10) / totalWeighting);* This delays the division until after multiplication, minimizing rounding errors.
Comments / Resolution	Fixed - Implemented recommended mitigation