BAIL security

Hyperdrive Tokenization

FINAL REPORT

February ‘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 - Tokenization
Website	hyperdrive.fi
Language	Solidity
Methods	Manual Analysis
Github repository	https://github.com/ambitfi/hyperdrive-contracts/tree/6399d8dfba060359646d0025955c39ec9723f991/packages/tokenization/contracts/protocol
Resolution 1	

2. Detection Overview

Severity	Found	Resolved	Partially Resolved	Acknowledged (no change made)	Failed resolution
High	11				
Medium	10				
Low	12				
Informational	10				
Governance	1				
Total	44				

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

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 USDC amount in the system and covers all edge-cases
- L1 HLP Vault has only withdraw lock as safeguard, no other measurements which could result in DoS
- All system calls are using the correct function selectors. We have no possibility to validate this (L1Write; PrecompileLib)
- Proxy contract on EVM is existent on L1
- SYSTEM_ADDRESS is the bridge contract
- transfer to SYSTEM_ADDRESS bridges funds to L1
- The EVM reads from the L1 state at the last block at the time of block production for EVM
- The EVM writes to the L1 state for the next following block at the time of block production for EVM
- The L1 / EVM block execution is sequentially. It can never happen that the L1 produces a new block while the EVM is still executing a block. Same vice-versa, the EVM can never produce a block while the L1 is executing a block.
- readSpot returns USDC amount with 8 decimals
- readEquity returns USDC amount with 6 decimals

- readWithdrawable returns USDC amount with 6 decimals

- Any balance on the EVM change is denominated with 8 decimals

- Writing to L1 block must be incorporated in the next L1 block execution

- During a deposit on L1, the funds flow will be as follows: SPOT -> PERP -> EQUITY

- During a withdraw from L1, the funds flow will be as follows: EQUITY -> PERP -> SPOT

- During a PERP -> SPOT withdrawal, it will withdraw from PERP to SPOT during block 1 and will then be available to bridge to EVM during block 2 which makes it available on EVM during subsequent block (block 3)

- Since funds are only sitting in the L1 SPOT balance after they have been withdrawn from the L1 PERP balance, they will always have the last two digits zero-d out.

HLVaultLensUpgradeable

The HLVaultLensUpgradeable contract is a simple lens contract which is used for frontend purposes

Privileged Functions

- none

Issue_01	getVaultQuery and getProxyAssets does not incorporate bridge scenario
Severity	**Medium**
Description	The getVaultQuery and getProxyAssets functions both attempt to return balances in the system. However, both functions do not incorporate the edge-case where funds have just been moved from EVM to L1 in the current block, as they now won't be reflected in the EVM state and also not in the L1 state.
Recommendations	Consider incorporating that scenario.
Comments / Resolution	

ProxyLib

The ProxyLib contract is a simple helper library which is used by the HLVaultUpgradeable contract in an effort to interact with the corresponding proxy contracts. It basically allows the HLVaultUpgradeable contract to interact with the Proxy contract as well as with the L1 view/read state.

Summarized, it allows for the following interactions:

a) Reading the SPOT balance from a proxy on L1 (8 decimals)
b) Reading the PERP balance from a proxy on L1 (6 decimals)
c) Reading the EQUITY balance from a proxy on L1 (8 decimals)
d) Withdrawing from the EQUITY balance for a proxy on the L1
e) Withdrawing from the PERP balance for a proxy on the L1
f) Withdrawing from a SPOT balance for a proxy on the L1
g) Depositing funds from the vault to EQUITY for a proxy on the L1

The system contracts are the following:

SYSTEM_WRITE = 0x3333333333333333333333333333333333333333

Allows to interact with the L1 which is reflected in the subsequent block

SYSTEM_ADDRESS = 0x2222222222222222222222222222222222222222

Allows to read from the L1 state of the previous block

The Proxy contract itself is not part of the audit and it is expected that all executions work correctly without side-effects.

Appendix: Deposit Flow

The depositEquity function allows for transferring funds from the HLVaultUpgradeable contract directly to the EQUITY for a proxy on the L1. This is done as follows:

a) Approve the proxy to spend USDC from the HLVault

b) Proxy calls transferFrom on USDC to transfer funds from HLVault in
c) Proxy transfers USDC to bridge which grants the Proxy address on the L1 chain funds upon next block execution
d) Execute a L1 write to convert SPOT funds to PERP funds
e) Execute a L1 write to convert PERP funds to EQUITY funds

Core Invariants:

INV 1: All L1 writes must be executed immediately in the subsequent L1 block

INV 2: Even if balance is denominated with 8 decimals, the last two digits always remain zero

INV 3: sendVaultTransfer = false must withdraw funds from EQUITY to PERP

INV 4: sendUsdClassTransfer = false must withdraw funds from PERP to SPOT

INV 5: sendSpot must bridge funds from L1 to EVM

INV 6: transfer to SYSTEM_ADDRESS must transfer funds from EVM to L1

INV 7: sendUsdClassTransfer = true must deposit funds from SPOT to PERP

INV 8: sendVaultTransfer = true must deposit funds from PERP to EQUITY

Privileged Functions

- none

No issues found.

USDMath

The USDMath contract is a simple library which converts a uint64 type to a custom USD type. It is used throughout the architecture in an effort to convert USDC with 8 decimals to USDC with 6 decimals and vice-versa. Moreover it exposes simple arithmetic operations such as add and sub for the custom USD type.

It is important to note that when funds are converted from 8 decimals to the USD type, the last two digits are inhernetly zero'd out.

Privileged Functions

- none

No issues found.

HLVaultUpgradeable

The HLVaultUpgradeable contract is a customized ERC7540 contract, which is basically an extension of the standard ERC4626 vault with an asynchronous withdrawal mechanism. Unlike the standard ERC7540 contract, the deposit mechanism is not asynchronous but straightforward.

The rationale behind implementing asynchronous withdrawals lies in the fact that funds are usually not sitting within the vault but are instead deposited on the L1 Hyperdrive chain in the HLPVault. This means there is no possibility to immediately honor withdrawals because funds will not be available during the execution block and are only available in the subsequent blocks due to the nature of the L1 withdrawal mechanism.

The contract is tied to the WithdrawalQueueUpgradeable contract which is basically a “plug-in” priority queue which allows users to request withdrawals in a priority based-manner depending on the provided feeBPS (between 1 and 10000) whereas the requestId with the lowest number is prioritized first. This mechanism will be explained in-depth within the “Priority Mechanism” appendix in the WithdrawQueueUpgradeable contract. It is not recommended to ever change the queue address as this will have inherent side-effects.

The withdrawal flow is split into three dedicated parts:

a) Request intention via requestRedeem
b) Request honoring via execute within the WithdrawQueueUpgradeable
c) Request redemption via withdraw or redeem

Parts a) and c) are handled within this contract while part b) is handled via the WithdrawQueueUpgradeable contract. It is also possible to cancel a request intent as long as it has not yet been executed. This will then simply allow the controller to reclaim the previously provided shares as if nothing has happened via the claimCancelRedeemRequest function.

Moreover, governance can add as many proxies as desired which then will be used to deposit into the HLPVault on the L1. The reason behind the incorporation of proxy contracts is that the HLPVault has a withdrawal lockup of 4 hours after each deposit, which means if there would only be one deposit contract, it would remain permanently locked and thus render it impossible to withdraw funds.

The contract also exposes a delegation mechanism which will be described in-depth in the corresponding appendix.

Appendix: totalAssets

The totalAssets function is the most fundamental function as it represents the total USDC amount in the system which is used for various different functionalities and most importantly for the conversion of shares to assets and assets to shares. Under all circumstances this function must report the accurate USDC value in the system. Any deviation from the real value can result in a loss of funds. Summarized it aggregates the following balances:

a) Vault balance (EVM, minus funds for claimable redemptions)
b) Proxy balances (EVM)
c) Spot balances (L1)
d) Perp balances (L1)
e) Equity balances (L1)

Appendix: Bridging Mechanism

Within the totalAssets function, there is a specific case where funds may not be accurately represented. This is whenever a deposit has happened which has triggered a cross-chain-transfer as within this exact block, funds are not sitting in the EVM chain and the L1 state is from the previous block, which means bridged funds are not yet reflected on the L1 state but also not anymore within the vault's balance. To solve this problem, a depositAccounting struct has been implemented which tracks the block.timestamp, block.number and amount. If now the totalAssets function is consulted while at the same time one or more deposits have happened which bridged funds to the L1, these funds will be incorporated into the totalAssets function.

Appendix: Delegation Mechanism

The contract exposes a delegation mechanism which allows an address to determine one or multiple operators on behalf of it via the setOperator function. Approved operators do the following actions:

a) Withdraw redemption-ready requests
b) Redeem redemption-ready requests

c) Cancel redeem requests
d) Claim cancelled redeem requests

Appendix: Request Intent and Cancellation

The first step of a withdrawal/redemption in this asynchronous vault is the intent expression of a redemption. This can be done via the requestRedeem function which transfers in shares but does not yet burn them. It then pushes the request into the priority queue and assigns the corresponding requestId to the user. The user has the possibility to cancel this request at any time but only until it has been executed. The cancellation will simply mark the requestId as cancelled and gives the user the right to reclaim the provided shares. After a cancellation it is as if no intent expression has happened.

Appendix: Deposit Flow

The contract allows users to deposit via two dedicated mechanisms:

mint: This function allows a user to specify a desired amount of shares and then transfers the corresponding amount of assets in. It requires the user to not only provide the asset amount for the corresponding shares but also the asset amount for the corresponding fees.

The math for this is as follows:

a) Calculate corresponding amount of assets for shares:
> assets = shares * totalShares / totalAssets

b) Apply the fee on top of the assets amount:
> fee = assets * feeBPS / 10_000

c) Calculate the shares for the corresponding fee amount:
> feeShares = (assets * totalSupply / totalAssets) - shares

deposit: This function allows a user to specify the asset amount which the user wants to provide and calculates the corresponding amount of shares. It is expected that the asset parameter already includes the fees in it for the proper share calculation.

The math for this is as follows:

a) Calculate the fee amount from the provided assets:
> fee = assets * feeBPS / (feeBPS + 10_000)

b) Calculate the share amount the user receives for asset deducted by the fee amount
> (assets - fee) * totalShares / totalAssets

c) Calculate the shares for the corresponding fee amount:
> (assets * totalSupply / totalAssets) - shares

The fee is configurable and always minted to address(this) and can then be retrieved by governance via the claimReserves function.

Appendix: Withdraw Flow

The most sophisticated part of this contract is the withdrawal flow, as it is no synchronous process like with standard vaults but an asynchronous process which honors requests based on the priority queue.

Furthermore, most of the time during the regular business logic, all funds are staked on the HLPVault on the L1 chain which requires a three-step execution until funds are finally sitting within the HLVaultUpgradeable contract to finalize a redemption.

The standard withdrawal flow including withdrawing equity from the HLPVault is as follows:

a) User expresses the intent of a redemption via the requestRedeem function. Shares will be transferred from the user to the vault but not yet burned and a requestId is created which is based on the provided priority fee (feeBPS) among some other factors which will be explained in the corresponding appendix and pushed into the queue.

b) The requestId is honored via the execute function within the WithdrawQueueUpgradeable contract which triggers prepareRequestRedeem on the vault to withdraw the PERP + SPOT balance from all proxies on the L1 to the EVM and eventually triggers a withdrawal from the EQUITY in case funds on EVM + PERP + SPOT is insufficient to honor the request. If funds on the EVM chain are sufficient to honor the

expected assets, the finalizeRequestRedeem function is called which burns the shares for the corresponding request and increases claimable assets for the user

c) The user can finally claim his assets via the withdraw or redeem function

There are a couple of different transition scenarios for the execute function which are mainly related to the possibility that a request may not be fully covered by the current EVM or current idle L1 balance. We have identified the following different scenarios:

a) A request is executed and the EVM balance is sufficient to honor the request, no withdrawal from L1.spot, L1.perp or L1.equity is necessary

b) A request is executed and the EVM balance is insufficient to honor the request and the idle L1 balance is sufficient to cover the leftover amount. The necessary leftover amount is withdrawn from L1.spot and/or L1.perp which allows for the request to be honored in the next subsequent block

c) A request is executed and the EVM balance is insufficient to honor the request and the idle L1 balance is insufficient to cover the leftover amount. The necessary leftover amount is withdrawn from L1.equity to L1.perp which allows for the withdrawal from the idle L1 balance in the next block and finally the execution in the subsequent block. In total there are three different blocks necessary until the request is honored. **This is the standard scenario which will be executed, as most of the time there is no idle vault balance nor idle L1.spot/L1.perp balance.**

d) A request is executed and the EVM balance is insufficient to honor the request and the idle L1 balance is insufficient to cover the leftover amount. The necessary leftover amount is withdrawn from L1.equity to L1.perp. However, the unlocked L1.equity amount is insufficient which means it is necessary to wait until another proxy becomes unlocked. In the meantime, assets will be transferred to the EVM vault in the subsequent execute calls.

Core Invariants:

INV 1: totalAssets is denominated in 8 decimals

INV 2: prepareRequestRedeem must only be callable once per block

INV 3: asset[]).balanceOf(address(this)] - $.totalClaimableAssets must always be below 5 USDC

at any state besides in the specific edge-case where L1 funds are transferred to the vault but not yet finalized for redemption.

INV 4: Received funds from L1 must always have zero-d out the last two digits.

INV 5: totalClaimableShares and totalClaimableAssets must always have a ratio of 1

INV 6: finalizeWithdraw must result in the corresponding asset amount being withdrawn from all proxies to the vault balance

INV 7: finalizeRequestRedeem must never change the ER

INV 8: Before prepareRequestRedeem is called, funds from proxies on EVM must have been transferred to vault

INV 9: Within redeem and withdraw, the amount of shares must always exactly reflect the amount of assets, without any deviation

INV 10: totalAssets must incorporate crediting only if a deposit for the corresponding proxy has happened within the same block

INV 11: getWithdrawableL1 must represent only what has been withdrawn in the previous execute call from the L1 equity via withdrawEquity. Under no circumstances these should be more/less funds than this exact amount.

INV 12: Only the owner or operator can call: withdraw/redeem/cancelRedeemRequest/claimCancelRedeemRequest

INV 13: cancelRedeemRequest cannot be called if requestId has already been executed

INV 14: claimCancelRedeemRequest can only be called if cancelRedeemRequest has been called before

INV 15: Within withdrawL1, spot must always be zero

INV 16: getActiveProxy must return a valid proxy index

INV 17: convertToAssets must return the asset amount to a corresponding share amount with 8 decimals

INV 18: If no redemption is executed, funds will always sit within the HLPVault on the L1 chain (dust ignored)

INV 19: Each requestId must be unique under all circumstances

INV 20: The contract must only be used with USDC

INV 21: getWithdrawableL1 must always accurately return the amount which has been withdrawn from equity in the previous block

INV 22: readSpot must always return zero

Privileged Functions

- grantRole
- revokeRole
- setReserveFee
- setWithdrawQueue
- addProxy
- claimReserves

Issue_02	Governance Privilege: Full control by governance
Severity	**Governance**
Description	Currently, governance of this contract has several privileges for invoking certain functions that can drastically alter the contracts behavior. For example, there is no boundary for the proxy contracts to be added as well as governance has full control over roles which can result in a full loss of funds if ever compromised or malicious
Recommendations	Consider incorporating a Gnosis Multisignature contract as owner and ensuring that the Gnosis participants are trusted entities.
Comments / Resolution	

Issue_03	Permanent queue block by calling requestRedeem with controller = address(0)
Severity	**High**
Description	The requestRedeem function allows users to delegate their requested shares to any controller address. Currently there is no check that the controller is not address(0). Additionally, the execute function inherently assumes that if controller = address(0), the requestId is invalid (ie. there is currently no requestId in the queue): *(WithdrawQueueLib.QueueKey memory key, WithdrawQueueLib.QueueEntry memory entry) = $.queue.peek();* *if (key.controller == address(0)) {* *// nothing was found* *return;* *}*

	A sophisticated attacker can combine these two facts to permanently block the queue, by simply calling the requestRedeem function with controller = address(0), which will then result in a permanent early return of the execute function.
Recommendations	Consider not allowing the controller parameter to be address(0).
Comments / Resolution	

Issue_04	Possible flash-theft via L1 mempool inspection
Severity	**High**
Description	Currently, the totalAssets function reads SPOT + PERP + EQUITY from the last block on the L1. If there is a mempool on the L1 which can be observed, it is possible for malicious users to gauge which transactions are being executed which allows them to frontrun these executions with a deposit to then immediately experience an increase of the share value which allows for an immediate redemption with a profit.
Recommendations	There is no trivial fix for this issue as this is inherent existing by design due to the EVM <-> L1 interaction
Comments / Resolution	

Issue_05	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	

Issue_06	Queue can be DoS'd by executing multiple 1 wei withdrawals with large feeBPS
Severity	**High**
Description	Currently, any user can call the requestRedeem function with an amount as low as 1 share. This can be trivially abused by requesting thousands of 1 share redemptions with a high feeBPS which then results in these requests being pushed at the top of the queue before any other request. Depending on the willpower of the attacker and due to the once per block execution, this can grief the queue more or less indefinitely. Similar to that, a user can also request normal redemptions which simply result in the received amount of assets to be below 1e2 (DUST) which then cancels out the request within finalizeRequestRedeem. This means no fee will be applied in that scenario
Recommendations	Consider implementing a reasonable lower limit for redemption requests.
Comments / Resolution	

Issue_07	Queue DoS by iterative requestRedeem and cancelRedeemRequest call in same block
Severity	**High**
Description	Whenever a requestId is executed, the execute function will return early if the requestId has already been cancelled: *if (entry.status == WithdrawQueueLib.CANCELLED) {* *emit Execute(vault, key, WithdrawQueueLib.serializeKey(key), WithdrawQueueLib.CANCELLED, block.timestamp);* *return;* *}* This however also prevents further executions in the same block. An attacker can abuse this fact by simply requesting and cancelling in a loop which then blocks the queue.
Recommendations	Consider taking a nominal fee for each request.
Comments / Resolution	

Issue_08	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 110e18 assets - previewDeposit(110e18) - fee = assets * fee / (fee+10000) - 110e18 * 1000 / (11000) - fee = 10e18 - convertToShares(100e18) - convertToShares(100e18 , (1e18+1), 1) - assets * totalSupply / totalAssets - shares = 100e18 - deposit(alice, alice, 110e18, 100e18) - feeShares = convertToShares(assets, totalSupply, totalAssets) - shares - convertToShares(110e18, 1e18+1, 0) - feeShares = 110e18 - 100e18 - feeShares = 10e18 - totalSupply = 111e18 - totalAssets = 110e18 - alice supply = 100e18

	- shares * totalAssets / totalSupply - 100e18 * 110e18 / 111e18 - 99e18 In the example one can see that Alice unexpectedly donated assets to the unbacked shares and experienced a partial loss of funds.
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	

Issue_09	Certain execution states can not be honored due to the fact that proxy balances are withdrawn to the vault and can be compounded back to EQUITY before an execution is finalized
Severity	**High**
Description	Most of the time, there are not sufficient idle funds in the vault nor in L1.spot/perp to cover a redemption request. This means that equity is being withdrawn to the PERP balance which will then be withdrawn to the EVM balance. In an edge-case where the unlocked equity balance during the first execution is insufficient to cover the request, a second equity withdrawal is necessary. A problem however arises because the withdrawn equity funds are transferred from the proxy to the vault via the recover function during the 2nd subsequent execution. This then means these funds which are considered for request redemption are sitting idle in the vault and will most likely be recompounded back into the HLP vault before the leftover equity amount can be withdrawn. This can even result in a permanent block of the queue if the redemption amount is the majority of all assets in the system while

	at the same time these amounts are distributed among many different proxies. It will simply be an indefinite loop of equity withdrawal combined with subsequent re-compounding, resulting in an indefinite lock of the queue because a requestId which is already in the execution can also not be canceled.
Recommendations	A potential solution might be the removal of the recover execution, however, that means the whole withdrawal flow needs a full re-audit.
Comments / Resolution	

Issue_10	Queue can be DoS'd by calling execute with wrong vault address
Severity	**High**
Description	As already explained, the execute function can only be called once per block. This can be abused by a malicious party to simply call it with an incorrect vault address which then results in a DoS of the queue if called every block.
Recommendations	Consider simply using the queue only for one vault, there are also other parts of the code which prevent the queue from being used for multiple vaults.
Comments / Resolution	

Issue_11	Lack of slippage check during redemption
Severity	**Medium**
Description	The requestRedeem function allows a user to request a redemption which can then theoretically be fulfilled in the next block if there are no other requests in the queue and/or the feeBps value is high. There is currently no slippage check for the received output amount of assets, while it theoretically is possible that the exchange rate decreases until the transaction is written to the blockchain. This is also true for the execute function, however, due to the nature of the mechanism it is non-trivial to implement such a slippage check here.
Recommendations	Consider implementing a simple slippage check upon requestRedeem.
Comments / Resolution	

Issue_12	Insufficiently thought-out pausing mechanism
Severity	**Medium**
Description	The whenNotPaused modifier is applied upon several functions. However, the application seems to be insufficiently thought out since it is applied on certain spots where it should not be, such as: claimReserves and is missing for spots where it should be applied such as: requestRedeem cancelRedeemRequest claimCancelRedeemRequest
Recommendations	Consider rethinking about the application of the whenNotPaused modifier.
Comments / Resolution	

Issue_13	Once per block execution will never work with multiple vaults
Severity	**Medium**
Description	The queue is designed to work with multiple different vaults. However, due to the oncePerBlock modifier within the execute function, this will just never work for multiple vaults.
Recommendations	Consider shifting the oncePerBlock mechanism to each vault isolated.
Comments / Resolution	

Issue_14	Initial share minting without totalAssets will result in erroneous calculation of previewMint
Severity	Medium
Description	The contract mints a designated amount of shares upon the initialization. We have already explained in another issue that this is an invalid state which naturally never occurs for vaults. Besides of the fact that the first depositor donates assets, it will also result in the mint function not being callable due to an inconsistent edge-case within the convertToAssets function: *return totalSupply_ == 0 ? shares : shares.mulDiv(totalAssets_, totalSupply_, rounding);* This will result in an underflow within the feeShares calculation and potentially even worse issues based on the testing setup. **Illustrated:** Alice calls mint with 100e18 shares - previewMint(100e18) - assets = convertToAssets(shares, totalSupply +1, totalAssets +1) - assets = convertToAssets(100e18, 1e18+1, 1) - shares * totalAssets / totalSupply - 100e18 * 1 / (1e18+1) - assets = 99 - ignore fee for simplicity example - deposit(alice, alice, 99, 100e18) - feeShares = convertToShares(assets, totalSupply, totalAssets) - shares - convertToShares(99, 1e18+1, 0) - this returns shares = assets - feeShares = 99 - 100e18 - **UNDERFLOW revert**

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	

Issue_15	Lack of offset incorporation during feeShares calculation
Severity	**Medium**
Description	Currently, whenever the feeShares value is calculated, the offset is ignored: *uint256 feeShares = convertToShares(assets, totalSupply(), totalAssets(), Math.Rounding.Floor) - shares;* This will result in less feeShares than expected. This can furthermore result in an underflow in certain scenarios because the calculated share amount does not incorporate the offset and could thus become smaller than the **subtracted shared amount**: *convertToShares(assets, totalSupply(), totalAssets(), Math.Rounding.Floor) - **shares**;* This is simply based on the fact that totalSupply does not incorporate the offset and thus is smaller than expected.
Recommendations	Consider incorporating the offset as well as following the totalAssets +1 approach.
Comments / Resolution	

Issue_16	Non-changeable depositLockup
Severity	Low
Description	The depositLockUp variable must always represent the lockup on the HLPVault to ensure users can withdraw from unlocked proxies. There is currently no way to change the depositLockUp. This will result in issues if this value is ever changed on the HLPVault directly.
Recommendations	Consider implementing a setter.
Comments / Resolution	

Issue_17	Lack of zero check for depositLockUp
Severity	Low
Description	Currently, there is no validation during the initialization that the depositLockUp is non-zero. If this value is set to zero, it will always revert due to a division by zero within the getActiveProxy contract
Recommendations	Consider validating the parameter accordingly.
Comments / Resolution	

Issue_18	Potential inconsistency with depositLockUp setting due to L1 state
Severity	Low
Description	The depositAccount[proxy].timestamp value is set whenever the deposit is triggered and signalizes that the corresponding proxy is now locked because a deposit to the HLPVault is happening. There may be inconsistency issues due to the fact that the deposit is only triggered in the subsequent L1 block and thus it may be possible that the proxy is already considered as unlocked while in fact it may be still locked.
Recommendations	Consider setting the depositLockUp value slightly higher to ensure such an inconsistency does not arise.
Comments / Resolution	

Issue_19	Violation of checks-effects-interactions pattern
Severity	Low
Description	Throughout the contract there are one or multiple spots which violate the checks-effects-interactions pattern, to ensure a protection against invalid states, all external calls should strictly be implemented after any checks and effects (state variable changes). claimableWithdraw claimCancelRedeemRequest ... In the claimCancelRedeemRequest this can even result in full draining of funds if a custom token is used.
Recommendations	Consider ensuring that only USDC will be used. If any other token will be used, a reentrancy guard must be applied.

Comments / Resolution	

Issue_20	Lack of reasonable fee setting
Severity	Low
Description	The setReserveFee 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 a reasonable validation.
Comments / Resolution	

Issue_21	Transfer-tax token incompatibility
Severity	Informational
Description	This contract is not compatible with transfer-tax tokens. If these token types are used for any purpose within the contract, this will result in down-stream issues and inherently break the accounting.
Recommendations	Consider not using these tokens.
Comments / Resolution	

Issue_22	Unnecessary complexity for transferFrom logic
Severity	Low
Description	The vault -> proxy transfer is following an approval and transferFrom mechanism. This mechanism is not redundant as a simple transfer is already sufficient.
Recommendations	Consider simplifying this logic.
Comments / Resolution	

Issue_23	Deposit allowance below MIN_VAULT_DEPOSIT
Severity	Low
Description	The deposit function allows to deposit an amount which is below MIN_VAULT_DEPOSIT. The only side-effect from that is that this amount is not compounded into the equity position. However, there is no real legitimate reason for why it should be allowed for users to deposit an amount below MIN_VAULT_DEPOSIT. 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.
Recommendations	Consider preventing deposits below MIN_VAULT_DEPOSIT.
Comments / Resolution	

Issue_24	Potential of faulty states via duplicated requestId
Severity	**Low**
Description	The enqueue logic explicitly prevents the addition of duplicated requestIds. However, there is at least one scenario which allows for duplicating a requestId. Specifically this is if a requestId was created and executed in the same block as it is not existent in the queue anymore but still remains in the "current" storage of the queue. If now at the same block the exact same request is created again, the same requestId which is sitting in current will be pushed into the queue. Using this technique, it is possible to create the same requestId two times. Another scenario would be the creation, cancellation, execution and re-creation in the same block which essentially creates a requestId which has previously been created but now canceled While we could not find any tied malicious exploit possibility from that faulty state, we highly recommend preventing it.
Recommendations	Consider implementing an isUsed mapping which simply marks a requestId as used and does not allow for the creation of a used requestId.
Comments / Resolution	

Issue_25	Incorrect event emission for the requestRedeem function
Severity	**Informational**
Description	The requestRedeem function emits the following event: *emit RedeemRequest(controller, owner, requestId, msg.sender, shares);* The owner will always be the owner address, no matter whether funds are transferred from the msg.sender or the owner.
Recommendations	Consider setting the owner address to the actual address where funds are transferred from.
Comments / Resolution	

Issue_26	claimableRedeemRequest returns aggregated value for all requestIds from controller
Severity	**Informational**
Description	The claimableRedeemRequest function allows for a requestId parameter but then requires this requestId to be zero and does not further use this parameter for any logic related purpose. The function then returns the aggregated value for all requestIds from the controller.
Recommendations	Consider simply removing the requestId parameter
Comments / Resolution	

Issue_27	Violation of specification for withdrawal flow
Severity	Informational
Description	The withdrawal flow exposes the previewClaimableWIthdraw, previewClaimableRedeem functions. These functions would usually return the shares and assets, which they actually also do based on the vault's logic. However, for example maxRedeem returns claimable instead of the actual provided amount of shares by a user.
Recommendations	Consider if it is desired to refactor this logic. If yes, it will require a re-audit of the withdrawal mechanism as this will introduce a new consistent conversion process. In our opinion, this can and should be safely acknowledged.
Comments / Resolution	

Issue_28	pendingRedeemRequest does not check ties from requestId to controller
Severity	Informational
Description	The pendingRedeemRequest allows for a requestId parameter and a controller. If the requestId is zero, it simply fetches all pending shares from the controller. If it is non-zero, it fetches all pending shares from the requestId. However, it does not check whether the requestId is indeed connected to the controller.
Recommendations	Consider keeping this in mind.
Comments / Resolution	

Issue_29	finalizeWithdraw is always redundant
Severity	Informational
Description	The finalizeWithdraw function is called within the finalizeRequestRedeem function and attempts to withdraw the desired funds from proxies to the vault. This is however redundant as the previous recover call already transferred all funds to the vault. Additionally to this issue, the loop execution within getWithdrawable is redundant.
Recommendations	Consider keeping this in mind.
Comments / Resolution	

Issue_30	Off-by-one error within totalAssets
Severity	Informational
Description	The totalAssets function implements the following property in the loop: *require(i <= $.proxies.length(), IndexOutOfRange());* This is incorrect as the index should never be equal to proxies.length. There is however no side-effect from this issue due to the logical execution of the loop.
Recommendations	Consider fixing this off-by-one error.
Comments / Resolution	

Issue_31	Incompatibility with assets != 8 decimals
Severity	Informational
Description	The contract inherently assumes that the token has 6 and 8 decimals (based on the chain). This will never work if a token with different decimals is added.
Recommendations	Consider only using USDC as an asset.
Comments / Resolution	

Issue_32	Chain reorg side-effects
Severity	Informational
Description	In the scenario of a chain re-org, it can have unexpected side-effects where for example the L1 state write is already triggered while the EVM state is reorganized.
Recommendations	Consider keeping this potential scenario in mind.
Comments / Resolution	

Issue_33	Inflation attack risk if zero shares are initially minted
Severity	Informational
Description	The contract is vulnerable to the standard share inflation attack if no shares are minted during the initialization. Since the attack is already widely known, we refrain from writing a detailed step-by-step explanation.
Recommendations	Consider always minting shares and providing the underlying asset amount.
Comments / Resolution	

WithdrawQueueUpgradeable

The WithdrawQueueUpgradeable contract is a queue contract which is compatible with multiple different vault contracts as each vault contract owns its separate and isolated storage. The queue is based on a priority mechanism which creates requestIds based on a unique mechanism. This will be explained in-depth in the corresponding appendix.

The contract exposes a permissionless execute function which is callable once per block and attempts to honor a requestId by potentially triggering a withdrawal from the L1 position. In an effort to incentivize the permissionless execution, the contract exposes a fee mechanism which takes the priority fee and distributes it once per week among all addresses which have called the execute function.

This is handled based on a share system where each address increases its shares by 1 whenever the execute function is called and then distributes fees based on all existing shares proportionally to each address whenever the claimFees function is called.

Appendix: Priority Mechanism

As already explained, the queue employs a priority mechanism where requests are stored in a **min-heap**, meaning the **smallest** integer value is considered the **highest** priority and will be served first. To achieve this:

1. Convert QueueKey into a 256-bit Integer (serializeKey)

a) key.priority is placed in the top 8 bits (shift 248)

b) (type(uint16).max - key.fee) is placed next (shift 200), making a higher fee produce a smaller value so it sorts higher in the min-heap.
c) key.timestamp is next (shift 160), ensuring older requests are favored if priority and fee are the same.
d) key.controller (the address) occupies the final 160 bits.

2. Negating the Fee

By storing (type(uint16).max - key.fee), the contract effectively inverts the fee in the serialized integer. A larger fee becomes a smaller integer in the top bits, so it naturally rises to the top of the min-heap.

Once the integer is inserted, the root of the heap is always the smallest integer, i.e., highest-priority request.

Core Invariants:

INV 1: execute must remove the requestId from the queue

INV 2: After execute has been called, it must be impossible to cancel the requestId

INV 3: Cancel must not be allowed if a requestId has already been executed

INV 4: peek must return the first requestId in the queue based on the priority logic

INV 5: If a requestId is canceled, it will not be removed immediately from the queue. Instead, it will be removed once the execute function for the corresponding requestId is called

INV 6: Before prepareRequestRedeem is called, funds from proxies on EVM must have been transferred to vault

INV 7: claimFees must only be called with past epochs

INV 8: executeEpochOperation must always be called before executeCurrent

INV 9: After executeEpochOperation, there must be no funds sitting in EVM proxy contracts anymore

INV 10: Within enqueue, it must fetch the larger of both values [feeBPS; redemptionFee.fee]

INV 11: cancel and enqueue must only be callable by the corresponding vault

INV 12: execute is only callable once per block

INV 13: clearCurrent is only called if a requestId has been fully executed

INV 14: finalizeRequestRedeem must only be called if funds within the vaul are sufficient to honor the request

INV 15: finalizeRequestRedeem must transfer the exact fee amount into the WithdrawQueueUpgradeable contract

INV 16: current must be reset after a request has been finalized

Privileged Functions

- setMinimumRedemptionAmount
- setMinimumRedemptionFee
- setAccountPriority

Issue_34	Fees can be stolen my mimicking execute with a fake vault
Severity	**High**
Description	The contract exposes a priority mechanism based on a fee which is then distributed among all executors via the claimFees function, per epoch. Due to a sophisticated exploit mechanism which is based on providing a fake vault and interacting with the WithdrawQueueUpgradeable contract it is possible to steal all fees which are sitting in the contract. **Illustrated:** a) Alice deploys a vault contract which matches the corresponding interface and exposes asset = USDC b) Alice creates a request on the malicious vault which calls the enqueue function on this contract and stores this request

	c) Alice calls the execute function with the malicious vault address such that everything passes and the storage of totalFees is increased without the vault actually transferring in the fees: *$.feeEpochs[feeEpoch[]].totalFees += fee;* d) Alice now waits until the epoch is over and calls the claimFees function which transfers all USDC out which are sitting in the contract. This is possible because the storage is correctly increased without having a transfer in of the funds which then simply takes the legitimate fee amount from the correct vault's interactions.
Recommendations	Consider whitelisting allowed vault addresses. In the ideal scenario this will only be the USDC vault.
Comments / Resolution	

Issue_35	Withdrawals can be blocked by calling execute with wrong vault address
Severity	**High**
Description	Due to the oncePerBlock modifier, it is possible to trivially grief the execute function via a bot which simply calls execute once per block with a wrong vault address, effecteively DoS'ing the execution of legitimate requests.
Recommendations	Consider implementing a tie between oncePerBlock and the vault address.
Comments / Resolution	

Issue_36	Lack of access control for minimumRedemptionAmount
Severity	High
Description	The setMinimumRedemptionAmount function lacks an access control mechanism which allows any address to call it and set an arbitrary minimumRedemptionAmount which then can result in the finalizeRequestRedeem function entering the first condition and simply cancelling a request.
Recommendations	Consider implementing an access control mechanism for this function.
Comments / Resolution	

Issue_37	Contract is never compatible with multiple vaults due to oncePerBlock modifier
Severity	Medium
Description	Due to the fact that the oncePerBlock modifier is not vault based but global based, it will be impossible for the queue to be compatible with multiple vaults.
Recommendations	Consider implementing a tie between oncePerBlock and the vault address.
Comments / Resolution	

Issue_38	Fee upscaling within finalizeRequestRedeem can result in unexpected large fee for user
Severity	**Medium**
Description	Within the finalizeRequestRedeem function, fee is determined as follows: *uint256 fee = Math.min((amount * key.fee) / 10000, $.redemptionFee.maxAmount);* This means if a user executed a large redemption with a large feeBps, it can never be larger than maxAmount. Clever users can use this mechanism to provide a large feeBps and get a priority spot in the queue while still only paying maxAmount. This will become an issue if maxAmount is increased b/w request intent and request execution because this can now result in a loss for the user.
Recommendations	Consider notifying the community early about potential fee increases.
Comments / Resolution	

Issue_39	Fee upscaling within enqueue can result in unexpected large fee for user
Severity	**Medium**
Description	Within the enqueue function, feeBps is determined as follows: *feeBps = Math.max(feeBps, $.redemptionFee.fee).toUint16();* This can result in users unexpectedly paying a larger fee than assumed and can even be amplified if redemptionFee.fee has been changed shortly before the transaction execution.

Recommendations	Consider documenting this carefully on the frontend and notifying the community early about potential fee increases.
Comments / Resolution	

Issue_40	Lack of pausing functionality
Severity	**Medium**
Description	The contract inherits the PausableUpgradeable contract but does not actually use it. This can have disadvantages in emergency situations. Furthermore, important functions in the HLVaultUpgradeable contract are also not pausable which amplifies this issue.
Recommendations	Consider implementing the whenNotPaused modifier on desired functions.
Comments / Resolution	

Issue_41	Intermediate exchange rate state will be used to override amountOut
Severity	**Low**
Description	When execute is called, most of the time the request will not yet be fulfilled because the equity needs to be withdrawn first. In that scenario, it is necessary to call execute twice again after that until the request can be fulfilled. A problem arises in that scenario because during the first execute call, it will always override amountOut with the intermediate state: *entry.amountOut = Math.min(entry.amountOut,*

	IHLVault(vault).convertToAssets(entry.shares)); If now during the final execution, the exchange rate is actually higher than at the time of the first execute call, it will still use the exchange rate at this intermediate state, while the general idea is to use the smaller exchange rate between the time of the request initiation and final execution. This will then result in an unexpected loss for the user.
Recommendations	Consider if that indeed violates the design, if yes we recommend simply using the initial amountOut (which will then anyways be potentially reduced within prepareRequestRedeem.
Comments / Resolution	

Issue_42	Potentially extended execution process in edge-case
Severity	**Low**
Description	The execute function may need to be called multiple times in an edge-case where the first prepareRequestRedeem call results in amountOut < entry.amountOut and all subsequent calls result in amountOut < entry.amountOut AND amountOut > amountOut (from previous redeem call). This is due to the fact that the previous redeem call in that scenario always withdraws insufficient equity while at the same time the current amountOut is still smaller than entry.amountOut which results in a shortage of L1 funds and requires another trigger of execute until eventually: a) entry.amountOut < amountOut b) currentAmountOut < lastAmountOut But as long as currentAmountOut is larger than lastAmountOut while still being below entry.amountOut, further executions are

	necessary until the withdrawal is sufficient to cover the next amountOut value.
Recommendations	Consider keeping this scenario in mind. A possible solution would be setting entry.amountOut to amountOut if it is below entry.amountOut which then uses entry.amountOut in case amountOut for the next execute is higher. However, that would introduce further complexity and potential edge-cases. Thus we recommend simply keeping this issue in mind as it is very rare that multiple subsequent execute calls always result in a larger amountOut than then previous call.
Comments / Resolution	

Issue_43	Addresses within prioritySet can never be removed
Severity	**Low**
Description	The setAccountPriority function allows assigning a priority number to an address and adds the address to the priorityAccounts set. However, there is currently no possibility to remove an address from the set.
Recommendations	Consider implementing a simple function to remove an address from that set and setting back the default number to it. We do not recommend any more intrusive changes.
Comments / Resolution	

Issue_44	getExchangeRate function never works
Severity	Low
Description	The getExchangeRate function uses 1**decimals for the asset determination. While this should convert to 1e18 (or whatever decimal value) it will simply be 1: *function getExchangeRate(address vault) private view returns (uint256) {* *return IHLVault(vault).convertToAssets(1 ** IHLVault(vault).decimals());* *}*
Recommendations	Consider changing that to 10 ** decimals.
Comments / Resolution	

WithdrawQueueLib

The WithdrawQueueLib contract is a helper library which is used by the WithdrawQueueUpgradeable contract and exposes functionality for creating the requestId and adding/removing requests from the queue.

Core Invariants:

INV 1: The same requestId cannot be added twice

INV 2: enqueue must insert the requestId on the corresponding spot in the heap

INV 3: enqueue must map the entry to the requestId

INV 4: dequeue must pop the first requestId from the heap and remove the corresponding entry

INV 5: peek must return key and entry from the top requestId on the heap

Privileged Functions

- none

No issues found