BAIL security

Hyperdrive Tokenization

FINAL REPORT

July '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/26a3da3d2f53040911f7ef901addca61a8b18240/packages/tokenization/contracts/protocol
Resolution 1	

2. Detection Overview

Severity	Found	Resolved	Partially Resolved	Acknowledged (no change made)	Failed resolution	Open
High	2					
Medium	5					
Low	8					
Informational	6					
Governance	1					
Total						

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

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 contract is in charge of interacting with the hyperliquid system program to manage transfers to and from the hypercore. It contains a set of proxy contracts, which are cycled through over time and deposited to and withdrawn from. It deposits funds into the proxy account on hypercore, sends them to the perp balance and adds them in a vault. Redemptions are done in the exact opposite way.

Appendix: Proxies

Instead of depositing to a single account, the contract manages an array of proxies, and chooses one of them as the current active proxy, and deposits funds to that proxy. This is done because once funds are deposited into a vault, the depositor is not allowed to withdraw funds from that vault until a certain lock duration has passed. This design allows the controller to have some proxies in the lock duration while having other proxies free to redeem funds from.

Appendix: Deposits

During deposits, funds are taken from the TokenizedVaultUpgradable contract and sent to the current active proxy. These funds are then through a sequence of operations and end up in the vault.

1. Funds are sent to the system address for that token in order to deposit it to the account on hypercore.
2. A CoreWrite event is emitted to transfer the received funds on the hypercore from the spot to the perp balance.
3. A CoreWrite event is emitted to transfer the received perp funds into the vault.

Appendix: Redemptions

Redemptions follow a similar three-step process. Users can call the requestRedeem function to queue a redemption, which gets processed in batches.

1. A CoreWrite event is emitted to withdraw funds from the vault to the perp balance.
2. A CoreWrite event is emitted to withdraw funds from the perp to the spot balance.
3. Funds are sent to the system address, transferring them from hypercore to hyperEVM.

Once available in the hyperEVM, funds are sent to the TokenizedVaultUpgradable contract, waiting for direct redemptions from the user.

Appendix: Force actions

Admins can call the forceWithdrawEquity, forceWithdrawPerp, or forceWithdrawSpot function to interfere and enable redemptions at any step of the process in case of stuck funds.

Privileged Functions

- setMinimumWithdrawableEquity
- depositEquity
- withdrawEquity
- withdrawL1
- withdraw
- forceWithdrawEquity
- forceWithdrawPerp
- forceWithdrawSpot

Core Invariants:

INV 1: Deposits are processed on the current active proxy. Withdrawals cycle through the proxies starting from the proxy just after the current active one.
INV 2: getTotalAssets value must remain constant at every step of the deposit/redemption process, assuming no vault profits/losses.
INV 3: If there are vault withdrawals, it must be for at least minimumWithdrawableEquity amount.

Issue_01	Anyone can create a proxy contract, causing unbounded iterations
Severity	**High**
Description	In the CoreControllerUpgradable contract, the function for creating a proxy is not permissioned. Anyone can create a proxy, which will be added to the proxies set. Multiple functions like fetching the total assets in the vault rely on the proxies and iterate over them; thus, an attacker can simply create a huge amount of proxies, causing transactions to OOG on multiple important functions, such as requesting a redemption, as that internally calls convertToAssets.
Recommendations	Consider applying access control to the function for creating proxies.
Comments / Resolution	

Issue_02	Lockup period for proxies when depositing to HLP vault can lead to DOS of withdrawals
Severity	**High**
Description	Hyperliquid L1 enforces a lockup period for each deposit made to the HLP vault. As stated in the **documentation**: *"The deposit lock-up period is 4 days. This means you can withdraw 4 days after your most recent deposit. For example, if you deposit on Sep 14 at 08:00, you can withdraw on Sep 18 at 08:00."* **Code Insights**: • Script implementation: deposit lockup period set to 120 seconds • Tests: deposit lockup period set to 1 hour • Estimated: ~10 proxies will be deployed With frequent deposits from proxies, the lockup timer resets continuously, effectively preventing any withdrawals from the HLP vault. This can result in a Denial-of-Service (DoS) condition where no proxy is ever able to withdraw.
Recommendations	Set the deposit lockup period to 4 days.
Comments / Resolution	

Issue_03	No forceDeposit function can lead to unused funds sitting in the proxy
Severity	**Low**
Description	While the contract implements functions to force-withdraw via the various steps, it does not have a force-deposit function to utilize any free funds in the proxy, either on hyperEVM or hypercore. They are only utilized for processing withdrawals. If the vault incurs a loss during a redemption, the batch.assets will decrease, and so the entire amount of funds removed from the vault will not be used to pay the users. This will lead to some unused funds sitting in the proxy. This leads to capital inefficiencies, since those funds are not earning profits from the vault. However the impact is very low, since when processing withdrawals, these idle funds are used up first.
Recommendations	Consider implementing a forceDeposit function which can deposit back any idle funds either in the proxy or the spot or perp balance.
Comments / Resolution	

Issue_04	DEFAULT_ADMIN_ROLE is never set in the Proxy constructor
Severity	Low
Description	Contract Proxy is deployed from the createProxy function in the core controller. However, on deployment, although the execution role is granted to the core controller and the default admin role is made its role admin, there is no DEFAULT_ADMIN_ROLE assigned in the code. *constructor() {* *_setRoleAdmin(EXECUTION_ROLE, DEFAULT_ADMIN_ROLE);* *_grantRole(EXECUTION_ROLE, msg.sender);* *}* This does not pose any risks currently, but the default admin role functionality can be useful for configuring proxies later.
Recommendations	Consider assigning a DEFAULT_ADMIN_ROLE in the Proxy constructor.
Comments / Resolution	

Issue_05	Remove redundant Proxy import
Severity	Informational
Description	Contract CoreControllerUpgradeable imports the Proxy contract twice. *import { Proxy } from "@ambitlabs/hyperdrive-periphery-contracts/contracts/Proxy.sol";*
Recommendations	Remove the extra Proxy import
Comments / Resolution	

Issue_06	Remove redundant role admin assignments
Severity	**Informational**
Description	Contract CoreControllerUpgradeable sets the role admin for the ADMIN_ROLE, OPERATOR_ROLE, and WITHDRAWER_ROLE as seen in the snippet below. *_setRoleAdmin(ADMIN_ROLE, DEFAULT_ADMIN_ROLE);* *_setRoleAdmin(OPERATOR_ROLE, DEFAULT_ADMIN_ROLE);* *_setRoleAdmin(WITHDRAWER_ROLE, DEFAULT_ADMIN_ROLE);* However, this is not required as the DEFAULT_ADMIN_ROLE can grant/revoke roles by default for all roles in the contract. This is because DEFAULT_ADMIN_ROLE has a value of 0, which is also the default value for _roles[role].adminRole.
Recommendations	Remove the admin assignments
Comments / Resolution	

Issue_07	Require statements are missing error messages
Severity	**Informational**
Description	There are many instances of require statements not including any error messages; it is considered best practice to include an error message for each require statement.
Recommendations	Consider adding error messages.
Comments / Resolution	

Issue_08	Admin can drain all funds from the system
Severity	Governance
Description	The admin can grant themselves the OPERATOR_ROLE and call withdraw to drain all funds from the system.
Recommendations	Consider giving the operator role only to timelocked multisigs. This reduces the fallout in case of an admin private key leak.
Comments / Resolution	

TokenizedVaultUpgradeable

The TokenizedVaultUpgradeable contract handles the user operations and interacts internally with the CoreControllerUpgradeable contract to perform hypercore operations. The vault tokenizes user deposits and mints them ERC4626 shares in exchange for their funds. Users can queue redemptions, which burn their shares and pay them back their part of the vault assets.

Appendix: Deposits

Users interact with the vault similarly to ERC4626 vaults. Users are minted shares in exchange for their funds. Their funds are then sent off to the CoreControllerUpgradable contract, where they get deposited to hypercore.

Appendix: Redemptions

Redemptions are processed in batches lasting 30 seconds each. After this period, a batch is marked expired, and its funds can eventually be withdrawn from the hypercore. The contract first utilizes free funds in the proxy and its spot and perp balances in Hypercore. If insufficient to meet redemption requests, it withdraws from the vault.

Each batch is finalized with a designated number of shares and assets. Users participating in that batch can then claim a part of the batch assets, depending on the percentage of shares they burnt during the redemption process.

Appendix: execute

The execute function carries out a step in the redemption process. If funds are withdrawn, it tries to finalize the process by pulling funds from the proxies to the current contract. Otherwise, it sends requests to withdraw funds from the hypercore or the vault, as needed. Multiple execute operations are required for each complete redemption.

Core Invariants:

INV 1: Deposits should only work until maximumSupply cap is reached.

INV 2: totalAssets should not change at any internal step in the redemption or deposit process, assuming no vault profit/loss.

INV 3: batch.assets are locked during the first execute call, and can only decrease in subsequent calls.

Privileged Functions

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

Issue_09	Executing will fail when it shouldn't, as code ignores idle assets
Severity	**Medium**
Description	For a batch to be finalized, the assets that are immediately withdrawable from the proxy contracts must be sufficient to resolve the corresponding batch: *if (batch.assets > $.coreController.getWithdrawable()) {* *return false;* *}* However, further down in the execution flow, the code first withdraws the assets from the proxies and then tries to get any remaining assets from the idle balance: *uint256 balance = token.balanceOf(address(this)) - $.totalClaimableAssets;* *finalized += $.coreController.withdraw(address(this), assets);* *// take any remaining amount from the vault contract* *finalized += Math.min(assets - finalized, balance);* This causes a contradiction. Firstly, if we reach the code above, we would never resort to the idle balance, as we have already enforced that the proxies have enough assets. Secondly, even if the batch would have been confirmed normally, we would have returned early, as we ignored the idle balance in the initial check shown.
Recommendations	Include the idle balance in the initial check shown in the report.
Comments / Resolution	

Issue_10	Redemptions can get stuck due to 0-value withdrawals
Severity	**Medium**
Description	In the withdraw function, the contract tries to withdraw batch.assets amount of funds, where the assets amount has already been rounded off. So it will always have 2 zeroes in the end. However, the contract also compares this with the current available balances in the contract, which has no such constraints. If insufficient, it tries to withdraw the remaining, converted to 6 decimals. *equity = USDMath.min(equity, USDMath.fromWei(batch.assets - withdrawable - withdrawableL1))* The issue is that this can result in a 0 withdrawal attempt, since the fromWei function divides by 100. Say batch.assets is 1e8. Say the contract already has 99_999_998 in its withdrawable proxy balance. So it is missing 2 wei. So batch.assets - withdrawable evaluates to 2. fromWei of this number evaluates to 0. So the controller tries to unsuccessfully withdraw 0 wei from the perp. Thus, the withdrawal will perpetually fall into a failed withdrawal cycle until the admin comes and forces withdrawal of enough funds to prevent this trigger. Whenever batch.assets - withdrawable is less than 100 wei, this issue can trigger.
Recommendations	Consider rounding up the equity amount calculated with fromWei. This will ensure enough funds are withdrawn.
Comments / Resolution	

Issue_11	User can get locked out of redemptions if the vault gets almost completely liquidated
Severity	**Medium**
Description	When redeeming, there's a check that the user is withdrawing more than 0 assets in the redeem function. *require(assets > 0, ZeroAmountNotAllowed());* The issue is that this limit can get hit unintentionally if the vault gets almost completely liquidated. Say a vault has 100 million, so 1e16 assets (18 decimals). Say a user redeems the minimum amount of 1 USDC, 1e8 assets. Suddenly, the vault gets liquidated to a few dollars. Now, the user's redemption amount will evaluate to 0, so they won't be able to get any funds out. However, the main issue is that the user can still deposit more funds! Say they deposit 1000 USDC to the vault again, which then invests this into profitable positions, generating revenue again. However, since the user's redemption queue still has a 0 value withdrawal pending, they won't be able to withdraw funds from their new investments either. Once a user submits a redemption request, that request needs to be deleted via updateUserRedemption for future redemptions to flow through. This won't happen if the redemption value is 0. User-controlled perp vaults can be quite risky and frequently get liquidated completely. It's common to participate with 100s of USDC tokens, expecting either a 10x or a complete liquidation. Thus, in such vaults, this situation can be quite common, where a user gets queued for a 0 withdrawal batch because they were trying to exit the vault before a massively leveraged position gets liquidated, but failed to time it. Now the user can deposit again, participating in more trades, but they can never withdraw any of their newly deposited funds, since they will still have a 0 value withdrawal at the front of their redemption queue.
Recommendations	Consider allowing 0 amount withdrawals. Or a more restrictive check would be to only allow 0 value withdrawals if the user's entire batch results in 0, i.e. toAssets(batchRedemption, userRedemption.shares[requestId])=0.

	There needs to be some way implemented to dequeue 0 withdrawal batches so that they don't block withdrawal queues forever.
Comments / Resolution	

Issue_12	An attacker can request minimal redeems for any user to queue up their redemptions
Severity	**Medium**
Description	The requestRedeem function allows any msg.sender to request redeems for any controller (user account). The issue with this is that an attacker can perform these minimal redeem requests (whether they're finalized or remain pending after expiration), which would keep getting added to the user's request queue. Currently, the minimum redemption fee is 1 asset token as per the tests. This means the attacker has to supply 1 asset token per batch (created every 30s). Now, let's say the user (who has deposited heavily into the vault) decides to create a redeem request. This would add the request to the back of the queue. When the batch is finalized, the user would attempt to call redeem, hoping that it would simply give the user their assets. However, since the redeem function uses the first member from the queue, the user has to first clear all the attacker's redeem requests before being able to withdraw their assets. This is definitely not convenient for the user and would cost the user quite a lot of gas. More importantly, the user might refrain from depositing into the protocol again, which causes a loss to the protocol indirectly. To note, if the minimum redemption amount is set to 0, the user can just pass in 1 wei of share token to perform this attack more easily.
Recommendations	Consider not allowing any arbitrary controller address as the receiver of the funds. Only msg.sender should be marked as the controller.

Comments / Resolution	

Issue_13	Assets may never be deposited into the vault
Severity	Medium
Description	Below illustrates what happens when depositEquity is called. 1. Assets are transferred from the TokenizedVault to the CoreController 2. Assets are transferred into the hypercore spot balance 3. Assets are transferred into the perps balance 4. Assets are deposited into the vault Steps 1-3 will all be executed by the next block. Step 4 incurs a delay as deposits or withdrawals from the vault must be delayed for a few seconds before they are executed on hyperL1. Again, we know that only vault deposits and withdrawals are delayed; transfers to and from the spot and perps balance are not. This allows for a malicious user (or this could happen just by chance as well, from honest user redemptions) to force funds to stay in perps and never get deposited into the vault. This can be accomplished by backrunning a call to depositEquity with a call to execute, as we know the funds will sit in the perps balance for some time before being deposited. In the case where all of the funds are sitting in the perps balance, a user simply calls execute to pull any amount of funds from perps to spot balance, then when the vault deposit finally comes, there will not be enough funds to complete the deposit since a portion of the funds has been withdrawn from the perps balance. This will cause funds to sit idle and not accrue any interest.
Recommendations	Implementing a forceDeposit function to deposit unused funds into a vault fixes this issue. Admins can deposit unused funds into the vault.
Comments / Resolution	

Issue_14	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	

Issue_15	Non-standard operator check allows users to scam their operator
Severity	**Low**
Description	In the context of approvals, the owner of the funds trusts another user/contract to use his funds appropriately. However, the opposite is not true - the other user/contract does not necessarily trust the user who approved him, e.g., he might be some sort of a relayer-type actor who acts on behalf of the user to earn funds, but does not necessarily trust him. The current code allows the owner of the funds to scam their operator. When requesting a redemption, the following happens: *address from = isOperator(owner, msg.sender) ? owner : msg.sender;* *IERC20(address(this)).safeTransferFrom(from, address(this), shares);* If the sender is an operator of the owner's address, then the from address is the owner. If not, the from address is the sender. Then, the from address sends over the shares to the contract. This allows a malicious user to do the following:

	1. Approve an operator to redeem instead of him. 2. The operator has some share tokens, approved to the contract. 3. The operator redeems, however he gets frontrun by the user who removes the operator approval. Now, the from address is the sender who provides the shares. If the controller address provided is owned by the other user, then the operator got scammed and lost his shares to the attacker.
Recommendations	Consider reverting if the sender is not the owner and if the sender is not an operator of the owner.
Comments / Resolution	

Issue_16	exchangeRate is incorrect
Severity	**Low**
Description	The function for fetching the exchange rate is incorrect. It currently divides the total assets by the total shares, which would be correct if the vault worked with standard asset-to-share conversions. However, the contract is inheriting an OZ ERC4626 with a decimal offset, which changes the exchange rate to `(totalAssets + 1) / (totalSupply + 1)`.
Recommendations	Include the decimal offset in the calculation.
Comments / Resolution	

Issue_17	initialShares will accrue interest
Severity	Low
Description	When initialized, the vault mints some initial shares to itself, specified in the initialShares parameter. This is mainly done in an effort to prevent inflation attacks. The issue is that these shares are still valid in the system and will accrue value from the system, costing the other users. Say on deployment, 1000 initial shares were minted. Alice comes and deposits 10 assets, so she gets minted 10*1001/1=10010 shares. Now, if the assets grow to 20, Alice's shares now only represent 10010*21/11011=19 assets. The remaining 1 asset will be claimable by the dead shares. So even though Alice provided the entire up-front amount, she does not get the entire profit.
Recommendations	The choice for initialShares needs to be kept within a reasonable limit. Minting 100-1000 dead shares for tokens with 8 decimals will limit the impact on users.
Comments / Resolution	

Issue_18	No slippage on deposits/redeems
Severity	Low
Description	The contract uses an ERC4626 vault to mint shares representing user assets. The ratio of shares to assets is constantly changing, depending on the state of the vault. Since the user can only specify the amount of assets or the number of shares while depositing/minting, they can receive an unexpected amount of shares/assets if the vault exchange rate shifts drastically. Say a user was expecting 100 shares for 100 assets, but the vault does a large liquidation, and the exchange rate changes to 2. So the user will only get minted 50 shares for their 100 assets. This can lead to unexpected outcomes for the user.
Recommendations	Either implement a slippage system in this contract, or use a router to carry out user interactions and implement the slippage check in the router.
Comments / Resolution	

Issue_19	Vault does not support FOT asset token
Severity	Low
Description	Contract TokenizedVaultUpgradeable does not support fee-on-transfer tokens. This could lead to deposits/withdrawals failing if the asset is such a token. Currently, there are no vaults supporting such assets on hyperEVM, but if that were to change, this could be an issue in the future.
Recommendations	Due to the extremely low likelihood of FOT vault support, consider acknowledging this issue for now.
Comments / Resolution	

Issue_20	minimumRedemptionAmount can lead to stuck funds
Severity	Informational
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.
Recommendations	Consider allowing smaller value assets as long as they empty the user's account (remaining shares ==0).
Comments / Resolution	

Issue_21	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. We recommend removing this check since all batch.assets are given to the last redeemer because shares being redeemed will be exactly batch.shares.
Recommendations	
Comments / Resolution	

Issue_22	setRedemptionFee does not check fee > 10000
Severity	Informational
Description	The function setRedemptionFee allows the admin to set a fee higher than 100%(10,000).
Recommendations	Consider adding a check that ensures fee <10,000.
Comments / Resolution	