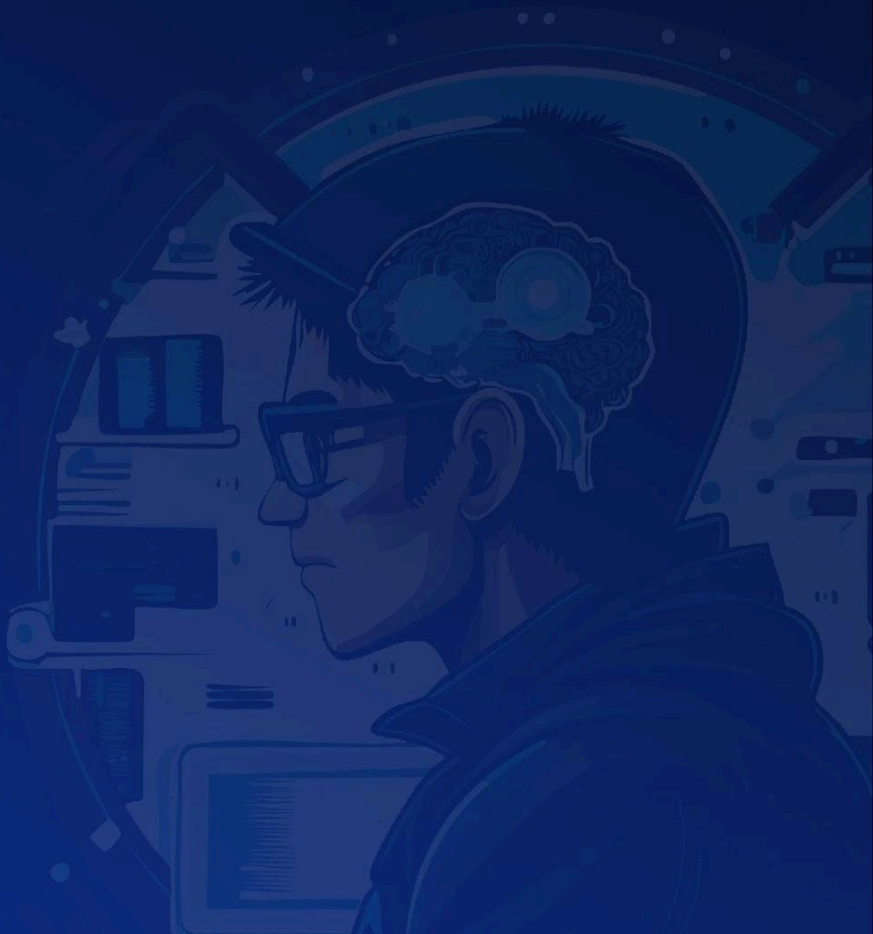
rHYPURR Vault

Security Review

Disclaimer

Security Review

rHYPURR Vault



Disclaimer

The ensuing audit offers no assertions or assurances about the code's security. It cannot be deemed an adequate judgment of the contract's correctness on its own. The authors of this audit present it solely as an informational exercise, reporting the thorough research involved in the secure development of the intended contracts, and make no material claims or guarantees regarding the contract's post-deployment operation. The authors of this report disclaim all liability for all kinds of potential consequences of the contract's deployment or use. Due to the possibility of human error occurring during the code's manual review process, we advise the client team to commission several independent audits in addition to a public bug bounty program.

Table of Contents

Security Review

rHYPURR Vault

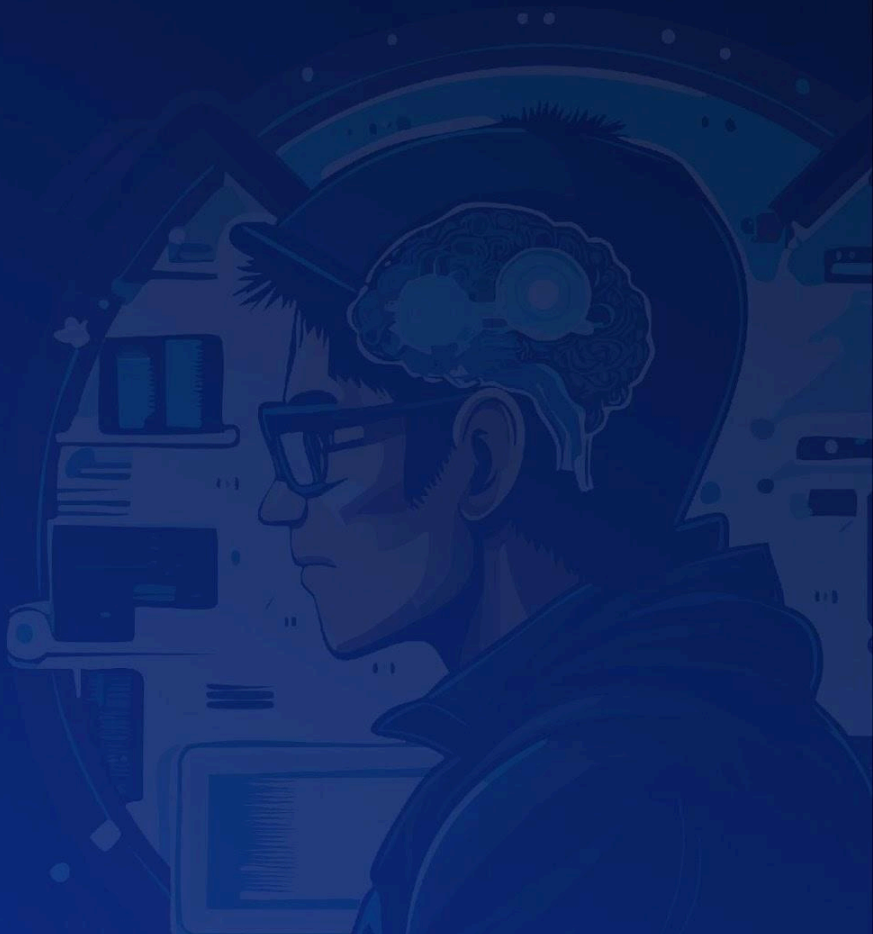
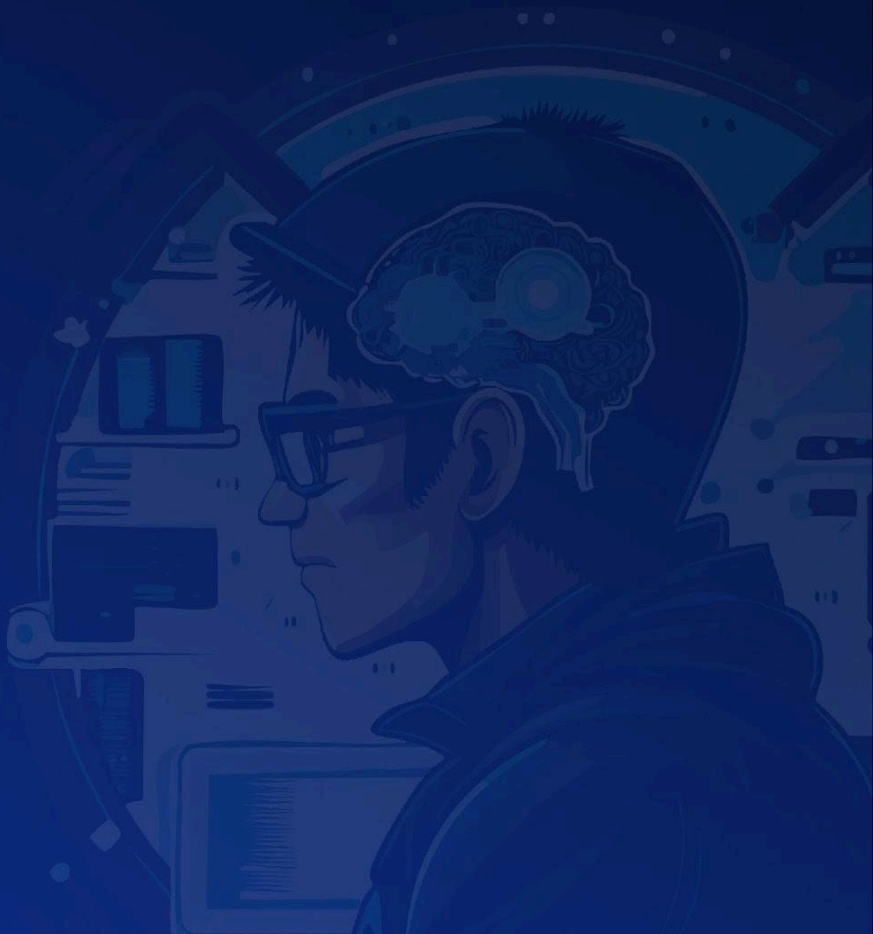


Table of Contents

Summary

Security Review

rHYPURR Vault



Summary

Three Sigma audited rHYPURR in a 1.2 person week engagement. The audit was conducted from 25/01/2026 to 27/01/2026.

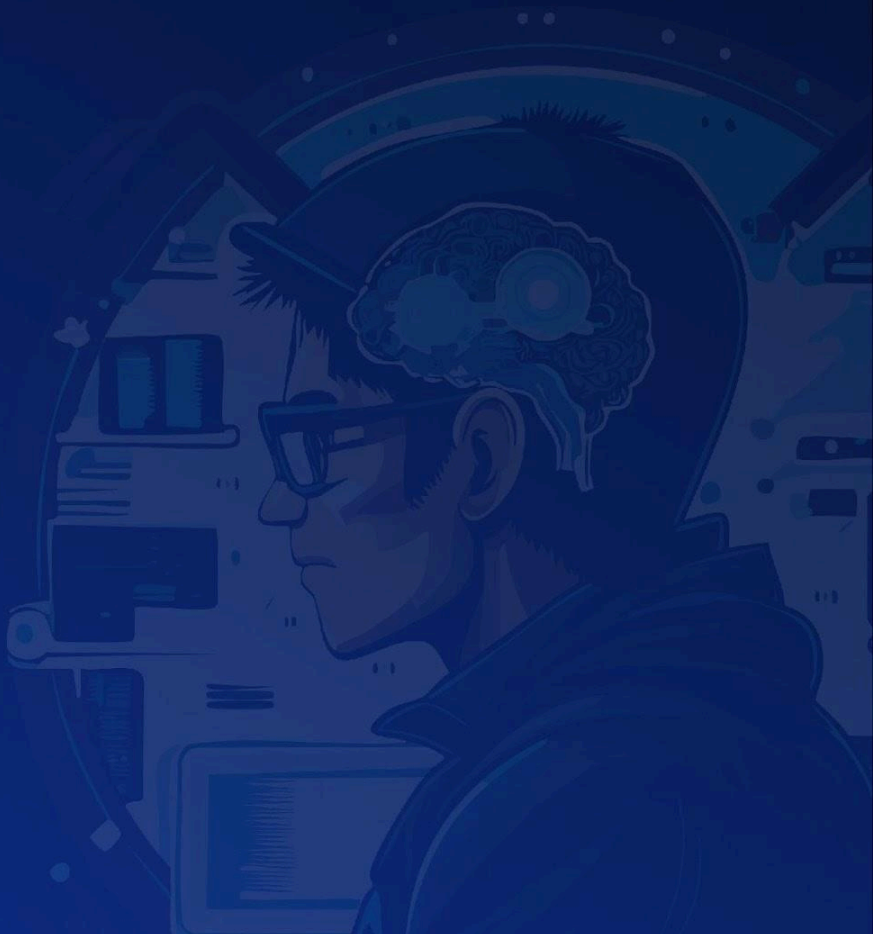
Protocol Description

rHYPURR is a tokenized ERC-4626 vault with ERC-7540 asynchronous deposit and redemption capabilities, deployed on HyperEVM (Hyperliquid's EVM sidechain). The protocol fractionalizes exposure to the Hypurr NFT collection and HYPE treasury reserves, allowing users to deposit HYPE and receive rHYPURR shares representing proportional ownership of the underlying assets.

Scope

Security Review

rHYPURR Vault



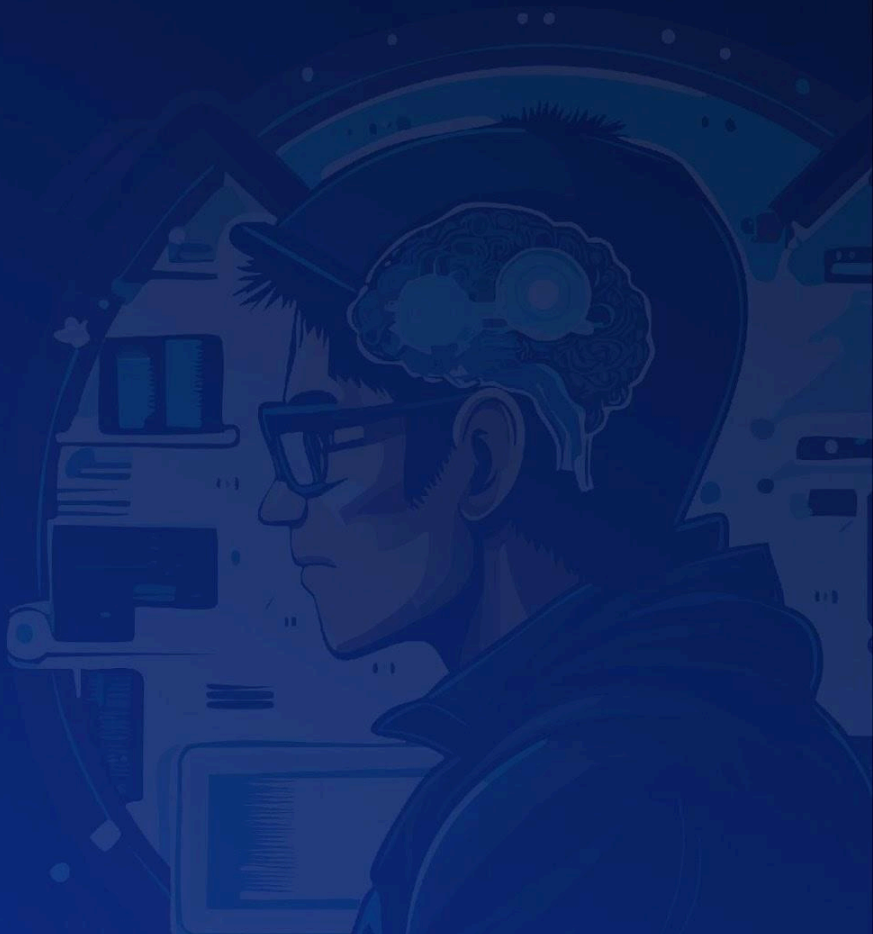
Scope

Filepath	nSLOC
contracts/rHYPURR.sol	699
Total	**699**

Methodology

Security Review

rHYPURR Vault



Methodology

To begin, we reasoned meticulously about the contract's business logic, checking security-critical features to ensure that there were no gaps in the business logic and/or inconsistencies between the aforementioned logic and the implementation. Second, we thoroughly examined the code for known security flaws and attack vectors. Finally, we discussed the most catastrophic situations with the team and reasoned backwards to ensure they are not reachable in any unintentional form.

Taxonomy

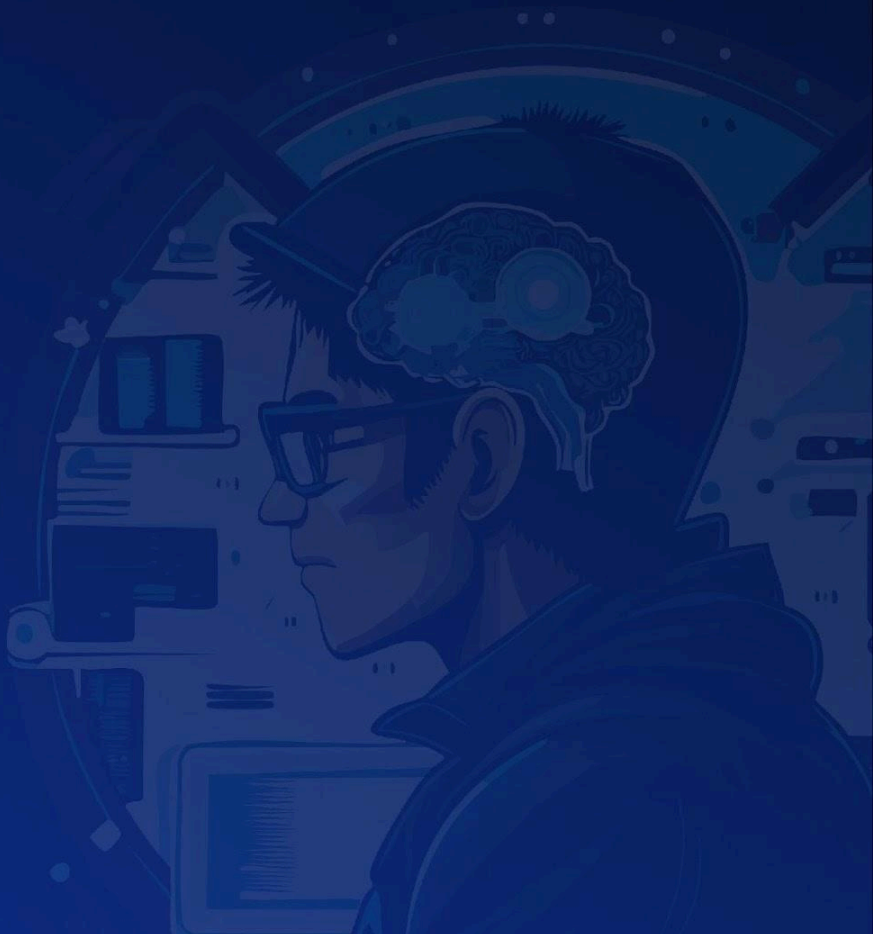
In this audit, we classify findings based on Immunefi's Vulnerability Severity Classification System (v2.3) as a guideline. The final classification considers both the potential impact of an issue, as defined in the referenced system, and its likelihood of being exploited. The following table summarizes the general expected classification according to impact and likelihood; however, each issue will be evaluated on a case-by-case basis and may not strictly follow it.

Impact / Likelihood	LOW	MEDIUM	HIGH
NONE	None		
LOW	Low		
MEDIUM	Low	Medium	Medium
HIGH	Medium	High	High
CRITICAL	High	Critical	Critical

Project Dashboard

Security Review

rHYPURR Vault



Project Dashboard

Application Summary

Name	rHYPURR
Repository	https://github.com/RebelDividends/rhypurr
Commit	c15e883
Language	Solidity
Platform	HyperEVM

Engagement Summary

Timeline	25/01/2026 to 27/01/2026
№ of Auditors	2
Review Time	1.2 person weeks

Vulnerability Summary

Issue Classification	Found	Addressed	Acknowledged
Critical	0	0	0
High	0	0	0
Medium	1	1	0
Low	2	1	1
None	2	2	0

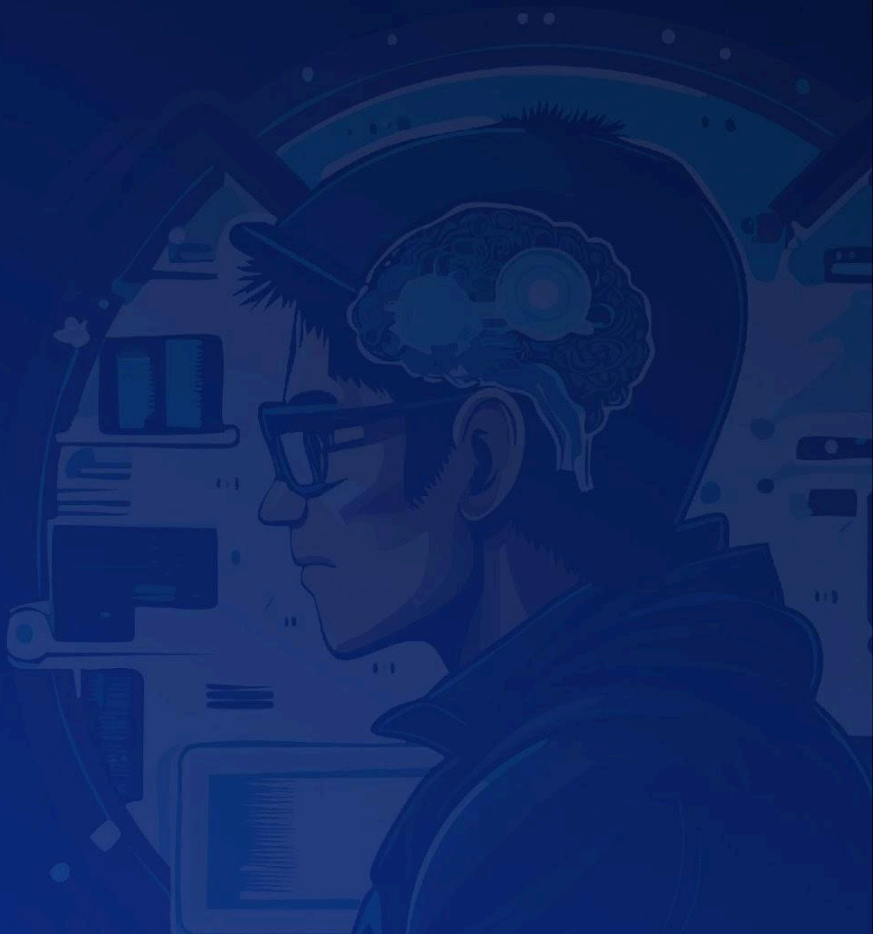
Category Breakdown

Suggestion	2
Documentation	0
Bug	3
Optimization	0
Good Code Practices	0

Risk Section

Security Review

rHYPURR Vault



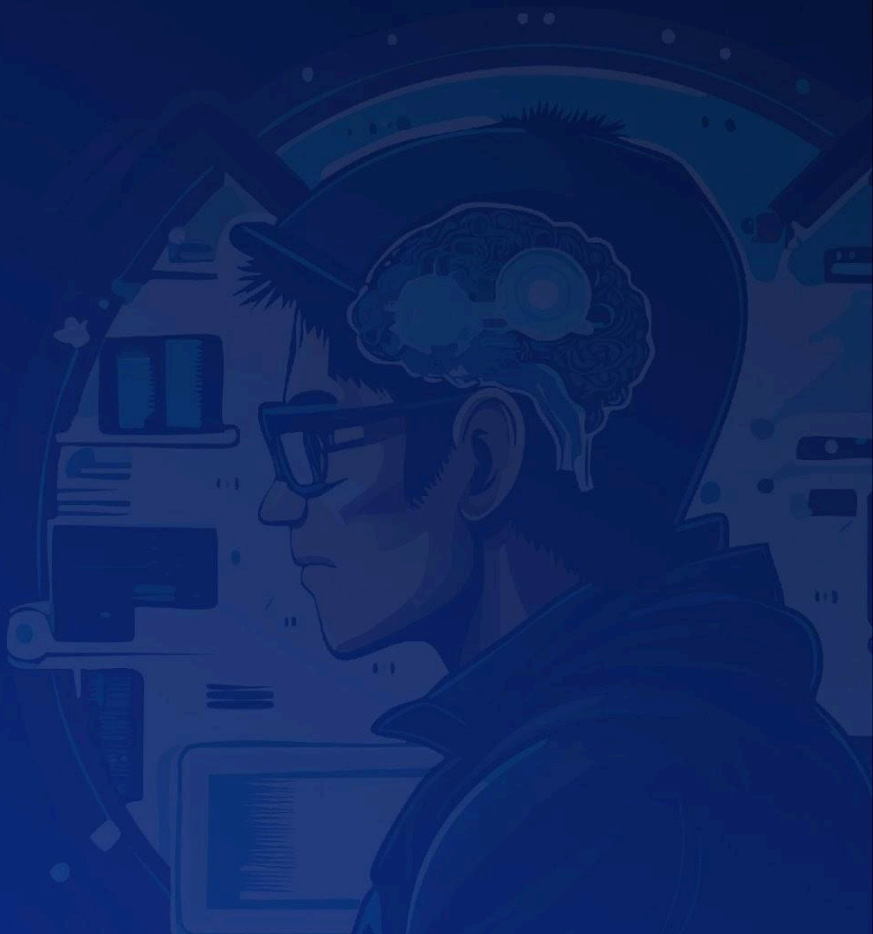
Risk Section

- **Administrative Key Management Risks**: The system makes use of administrative keys to manage critical operations. Compromise or misuse of these keys could result in unauthorized actions and financial losses.
- **Upgradability Risk**: Administrators have the capability to update contract logic at any moment due to the project's upgradable design. Contract upgradability, while beneficial, also poses the risk of harmful modifications if administrative privileges or upgrade processes are compromised.

Findings

Security Review

rHYPURR Vault



Findings

3S-Vault-M01

Reserved redemption liquidity accounting requires manual adjustment after processing redemptions

Id	3S-Vault-M01
Classification	Medium
Impact	Medium
Likelihood	High
Category	Bug
Status	Addressed in #2d0823d.

Description

The **rHYPURR** vault contract implements a liquidity management system that allows the treasury role to pre-reserve HYPE tokens for upcoming redemptions through the **rHYPURR::reserveRedemptionLiquidity** function. This reserved liquidity is tracked in the **reservedRedemptionLiquidity** state variable and is protected from automatic sweeps to the vault manager when excess liquidity is transferred.

The **rHYPURR::processRedemptions** function processes pending redemption requests by transferring HYPE tokens to users and burning their vault shares. However, the function does not automatically decrease the **reservedRedemptionLiquidity** counter when it uses reserved funds to fulfill redemptions. This creates an accounting discrepancy where the actual reserved liquidity decreases (through transfers) but the tracked amount remains unchanged.

The impact of this accounting mismatch is that the **rHYPURR::_sweepExcessLiquidity** function continues to treat already-spent funds as reserved. This prevents legitimate excess liquidity from being swept to the vault manager, as the calculation **protectedAmount = liquiditySweepThreshold + reservedRedemptionLiquidity** artificially inflates the protected amount. This reduces capital efficiency and requires the treasury role to manually call **rHYPURR::releaseRedemptionLiquidity** to correct the accounting.

Steps to Reproduce

1. Treasury role calls **rHYPURR::reserveRedemptionLiquidity** with 100 HYPE tokens to prepare for upcoming redemptions. State: **reservedRedemptionLiquidity = 100**.

2. User Alice requests a redemption for shares worth 100 HYPE tokens through **rHYPURR::requestRedemption**.

3. Oracle role calls **rHYPURR::processRedemptions** to fulfill Alice's redemption request. The contract transfers 100 HYPE to Alice and burns her shares. State: **reservedRedemptionLiquidity = 100** (unchanged).

4. New deposits arrive and the oracle processes them. When **rHYPURR::_sweepExcessLiquidity** executes, it protects 100 HYPE tokens that no longer exist in the contract, preventing a legitimate sweep of excess liquidity.

5. Treasury role must manually call **rHYPURR::releaseRedemptionLiquidity** with amount 100 to correct the accounting and allow subsequent sweeps to function properly.

Recommendation

Modify the **rHYPURR::processRedemptions** function to automatically decrease **reservedRedemptionLiquidity** when redemptions are fulfilled, up to the amount of reserved liquidity available. This ensures the accounting accurately reflects the actual reserved funds remaining in the contract.

```
function processRedemptions(
  uint256[] calldata requestIds,
  uint256[] calldata payoutAssets
) external override onlyRole(ORACLE_ROLE) whenNotPaused {
  if (requestIds.length != payoutAssets.length) revert InvalidRequest(0);
  uint256 head = _advanceRedemptionQueue(redemptionQueueHead);
  redemptionQueueHead = uint128(head);
  IERC20 hypeToken_ = IERC20(address(hypeToken));
  uint32 currentEpoch_ = currentEpoch;
+ uint256 totalPayouts = 0;
  for (uint256 i = 0; i < requestIds.length; ++i) {
    // ... existing validation and payout calculation ...
    if (fee != 0 && feeRecipient != address(0)) {
      hypeToken_.safeTransfer(feeRecipient, fee);
    }
    hypeToken_.safeTransfer(request.receiver, netAssets);
+   totalPayouts += payout;
    request.assetsPaid += netAssets;
    request.sharesPending -= sharesToBurn;
    pendingRedemptionShares -= sharesToBurn;
    _burn(address(this), sharesToBurn);
    bool completed = request.sharesPending == 0;
    emit RedemptionProcessed(requestId, request.owner, netAssets, fee, sharesToBurn,
```

```
completed);
    if (completed) {
      request.state = IRHYPURR.RequestState.Processed;
      unchecked {
        ++head;
      }
      head = _advanceRedemptionQueue(head);
    } else {
      break;
    }
  }
+ // Reduce reserved liquidity by the amount used for payouts
+ uint256 reservedRedemptionLiquidity_ = reservedRedemptionLiquidity;
+ if (reservedRedemptionLiquidity_ > 0) {
+   uint256 reduction = totalPayouts > reservedRedemptionLiquidity_
+     ? reservedRedemptionLiquidity_
+     : totalPayouts;
+   reservedRedemptionLiquidity = reservedRedemptionLiquidity_ - reduction;
+   emit RedemptionLiquidityReleased(reduction, reservedRedemptionLiquidity);
+ }
  redemptionQueueHead = uint128(head);
}
```

3S-Vault-L01

Liquidity sweep mechanism cannot operate when **liquiditySweepThreshold** is zero

Id	3S-Vault-L01
Classification	Low
Impact	Medium
Likelihood	Low
Category	Bug
Status	Acknowledged

Description

The **rHYPURR::_sweepExcessLiquidity** function is responsible for transferring excess HYPE tokens from the vault contract to the vault manager address. This mechanism ensures that the vault maintains only the necessary liquidity for processing redemptions while allowing the vault manager to deploy excess funds for NFT purchases or other treasury operations.

The function calculates excess liquidity by comparing the vault's HYPE balance against two protected amounts: the **liquiditySweepThreshold** (a configurable minimum balance to maintain) and **reservedRedemptionLiquidity** (funds reserved for pending redemptions). Any balance exceeding these protected amounts is transferred to the vault manager.

However, the function contains an early return statement when **liquiditySweepThreshold** is set to zero. This prevents the sweep mechanism from operating entirely, even though the vault may still hold excess liquidity beyond what is needed for **reservedRedemptionLiquidity**. The current implementation treats a zero threshold as a signal to disable sweeping, rather than as an instruction to sweep all funds except those reserved for redemptions.

Recommendation

Remove the early return when **liquiditySweepThreshold** is zero to allow the sweep mechanism to operate based solely on **reservedRedemptionLiquidity**. This enables administrators to configure a zero threshold when they want the vault to maintain only the minimum liquidity required for pending redemptions:

```
function _sweepExcessLiquidity() internal {
```

```
-   if (liquiditySweepThreshold == 0) return;
-
    IERC20Metadata hypeToken_ = hypeToken;
    uint256 balance = hypeToken_.balanceOf(address(this));
    // Protect both threshold and reserved liquidity from sweep
    uint256 protectedAmount = liquiditySweepThreshold + reservedRedemptionLiquidity;
    if (balance > protectedAmount) {
      uint256 excess = balance - protectedAmount;
      address vaultManager_ = vaultManager;
      IERC20(address(hypeToken_)).safeTransfer(vaultManager_, excess);
      emit ExcessLiquiditySwept(vaultManager_, excess);
    }
  }
```

Alternatively, if the current behavior is intentional, document that setting **liquiditySweepThreshold** to zero disables the automatic sweep mechanism entirely.

3S-Vault-L02

Fee collection during processing causes revert with low liquidity threshold and exposes users to unexpected fee increases

Id	3S-Vault-L02
Classification	Low
Impact	Medium
Likelihood	Low
Category	Bug
Status	Addressed in #8cf024a.

Description

The **rHYPURR::processDeposits** and **rHYPURR::processRedemptions** functions process pending requests by minting/burning vault shares and collecting protocol fees. Both functions calculate fees during processing rather than at request time, which creates two distinct impacts that affect both deposits and redemptions:

Impact 1: Transaction revert due to insufficient balance

When the **liquiditySweepThreshold** is set to a low value, the contract may not retain sufficient HYPE balance to cover fee transfers during deposit processing, causing the transaction to revert. When users request deposits via **_requestDeposit**, the function calls **_sweepExcessLiquidity** which transfers any HYPE balance exceeding **liquiditySweepThreshold + reservedRedemptionLiquidity** to the vault manager. This sweep occurs before fees are collected, potentially leaving insufficient funds in the contract to pay fees when deposits are later processed.

Impact 2: Users charged higher fees than expected

Charging fees at processing time rather than request time exposes users to fee changes between request and execution. A user who requests a deposit when the deposit fee is 0.5% may be charged 1% if the fee configuration is updated by the **TREASURY_ROLE** before their request is processed by the oracle. Similarly, a user who requests a redemption expecting a 0.5% redemption fee may be charged 1% if the fee is increased before processing. This violates user expectations and creates an unfair scenario where users cannot predict the actual cost of their transaction at the time they commit funds or shares.

Steps to reproduce (Impact 1)

1. Set **liquiditySweepThreshold** to 10e18 HYPE

2. Set **reservedRedemptionLiquidity** to 0 (no pending redemptions)

3. Set deposit fee to 1% (100 basis points)

4. User requests a deposit of 2,000e18 HYPE via **requestDeposit**

5. The **_sweepExcessLiquidity** function executes, transferring 1,990e18 HYPE to the vault manager (keeping only 10e18 in the contract)

6. Oracle calls **processDeposits** to process the request

7. Function calculates fee as 2,000e18 * 1% = 20e18 HYPE

8. Function attempts to transfer 20e18 HYPE to **feeRecipient**

9. Transaction reverts because the contract only holds 10e18 HYPE

Steps to reproduce (Impact 2)

1. Alice requests a deposit of 1,000e18 HYPE when the current deposit fee is 0.5% (50 basis points)

2. Alice expects to pay 5e18 HYPE in fees

3. Before the oracle processes Alice's request, the **TREASURY_ROLE** calls **setFeeConfiguration** to update the deposit fee to 1% (100 basis points)

4. Oracle calls **processDeposits** to process Alice's request

5. Alice is charged 10e18 HYPE in fees (1% of 1,000e18) instead of the expected 5e18 HYPE

Recommendation

Collect deposit and redemption fees during the request phase rather than the processing phase. This ensures fees are deducted before any liquidity sweeps occur and locks in the fee rate at request time, providing users with predictable fee expectations and preventing transaction reverts.

Modify **_requestDeposit** to calculate and deduct fees immediately:

```
function _requestDeposit(
  address owner,
  address receiver,
  uint256 assets,
  bool isNative
) internal returns (uint256 requestId) {
  _requireReceiverAndAmount(receiver, assets);
  _requireMinDeposit(assets);
  _requireDepositCap(assets);
```

```
    _requireNavFresh();
+   // Calculate and collect deposit fee at request time
+   uint256 fee = (assets * feeConfig.depositFeeBps) / BPS_DENOMINATOR;
+   uint256 netAssets = assets - fee;
+   if (netAssets == 0) revert InvalidRequest(0);
    requestId = nextDepositRequestId++;
    depositRequestEpoch[requestId] = currentEpoch;
    depositRequests[requestId] = IRHYPURR.DepositRequest({
      owner: owner,
      receiver: receiver,
-     assets: assets,
+     assets: netAssets,
      shares: 0,
      createdAt: uint64(block.timestamp),
      state: IRHYPURR.RequestState.Pending
    });
-   pendingDepositAssets += assets;
+   pendingDepositAssets += netAssets;
    if (!isNative) {
      IERC20(address(hypeToken)).safeTransferFrom(owner, address(this), assets);
    } else {
      IWHYPE(address(hypeToken)).deposit{value: assets}();
    }
+
+   // Transfer fee to recipient
+   if (fee != 0 && feeRecipient != address(0)) {
+     IERC20(address(hypeToken)).safeTransfer(feeRecipient, fee);
+   }
+
    emit DepositRequested(requestId, owner, receiver, assets);
    _sweepExcessLiquidity();
    return requestId;
  }
```

Update **processDeposits** to remove fee calculation:

```
  function processDeposits(uint256[] calldata requestIds) external override
onlyRole(ORACLE_ROLE) whenNotPaused {
    // ... existing validation code ...
    IRHYPURR.DepositRequest storage request = depositRequests[requestId];
    if (request.state != IRHYPURR.RequestState.Pending) {
      revert InvalidState(requestId, IRHYPURR.RequestState.Pending, request.state);
    }
```

```
-   uint256 assets = request.assets;
-   uint256 fee = (assets * feeConfig.depositFeeBps) / BPS_DENOMINATOR;
-   uint256 netAssets = assets - fee;
-   if (netAssets == 0) revert InvalidRequest(requestId);
+    uint256 netAssets = request.assets;
    uint256 sharesOut = Math.mulDiv(netAssets, ONE, epochNavPerShare);
    if (sharesOut == 0) revert InvalidRequest(requestId);
-   if (fee != 0 && feeRecipient != address(0)) {
-     hype.safeTransfer(feeRecipient, fee);
-   }
    request.shares = sharesOut;
    request.state = IRHYPURR.RequestState.Processed;
-   pendingDepositAssets -= assets;
-   depositCapCurrent += assets;
+    pendingDepositAssets -= netAssets;
+    depositCapCurrent += netAssets;
    _mint(request.receiver, sharesOut);
-   emit DepositProcessed(requestId, request.owner, netAssets, sharesOut, fee);
+    emit DepositProcessed(requestId, request.owner, netAssets, sharesOut, 0);
    // ... remaining code ...
  }
```

Modify **_requestRedemption** to store the fee rate at request time:

```
  function _requestRedemption(address owner, address receiver, uint256 shares) internal
returns (uint256 requestId) {
    _requireReceiverAndAmount(receiver, shares);
    _requireMinRedemption(shares);
    _requireNavFresh();
    requestId = nextRedemptionRequestId++;
    redemptionRequestEpoch[requestId] = currentEpoch;
    redemptionRequests[requestId] = IRHYPURR.RedemptionRequest({
      owner: owner,
      receiver: receiver,
      sharesOriginal: shares,
      sharesPending: shares,
      assetsPaid: 0,
      createdAt: uint64(block.timestamp),
      state: IRHYPURR.RequestState.Pending
    });
    pendingRedemptionShares += shares;
    _transfer(owner, address(this), shares);
    emit RedemptionRequested(requestId, owner, receiver, shares);
```

```
    return requestId;
  }
```

Update the **RedemptionRequest** struct to store the fee rate:

```
  struct RedemptionRequest {
    address owner;
    address receiver;
    uint256 sharesOriginal;
    uint256 sharesPending;
    uint256 assetsPaid;
    uint64 createdAt;
    RequestState state;
+   uint16 redemptionFeeBps;
  }
```

Store the fee rate in **_requestRedemption**:

```
    redemptionRequests[requestId] = IRHYPURR.RedemptionRequest({
      owner: owner,
      receiver: receiver,
      sharesOriginal: shares,
      sharesPending: shares,
      assetsPaid: 0,
      createdAt: uint64(block.timestamp),
      state: IRHYPURR.RequestState.Pending,
+     redemptionFeeBps: feeConfig.redemptionFeeBps
    });
```

Update **processRedemptions** to use the stored fee rate:

```
  function processRedemptions(
    uint256[] calldata requestIds,
    uint256[] calldata payoutAssets
  ) external override onlyRole(ORACLE_ROLE) whenNotPaused {
    // ... existing validation code ...
    IRHYPURR.RedemptionRequest storage request = redemptionRequests[requestId];
    uint256 sharesPending = request.sharesPending;
    uint256 maxPayout = Math.mulDiv(sharesPending, epochNavPerShare, ONE);
    uint256 payout = payoutAssets[i];
    if (payout < maxPayout) {
```

```
      uint256 minPartialPayout_ = minPartialPayout;
      if (minPartialPayout_ == 0) revert PartialRedemptionDisabled();
      if (payout < minPartialPayout_) revert PartialTooSmall(payout, minPartialPayout_);
    } else {
      payout = maxPayout;
    }
-   uint256 fee = (payout * feeConfig.redemptionFeeBps) / BPS_DENOMINATOR;
+    uint256 fee = (payout * request.redemptionFeeBps) / BPS_DENOMINATOR;
    uint256 netAssets = payout - fee;
    // ... remaining code ...
  }
```

3S-Vault-N01

Inaccurate NatSpec comment in **rHYPURR::processDeposits** regarding liquidity sweep behavior

Id	3S-Vault-N01
Classification	None
Category	Suggestion
Status	Addressed in #f0c4981.

Description

The **rHYPURR::processDeposits** function is responsible for processing pending deposit requests that have been queued by users during previous epochs. The oracle bot calls this function to mint **rHYPURR** shares to depositors based on the NAV snapshot from their request epoch. The function processes requests in FIFO order, applies deposit fees, and transfers the net deposited HYPE to users as tokenized vault shares.

The **@dev** NatSpec comment for this function states that it "auto-sweeps excess liquidity," which is inaccurate. The liquidity sweep operation occurs in the internal **_requestDeposit** function when users initially submit their deposit requests, not during the processing phase. Specifically, **_requestDeposit** calls **_sweepExcessLiquidity()** after receiving the user's HYPE tokens to transfer any vault balance exceeding the **liquiditySweepThreshold** to the Treasury Manager. The **processDeposits** function itself does not perform any liquidity sweep operations.

Recommendation

Update the NatSpec comment for **rHYPURR::processDeposits** to remove the reference to auto-sweeping excess liquidity:

```
 /**
  * @notice Process pending deposit requests for finalized epochs in FIFO order.
- * @dev Only callable by the oracle role; uses epoch NAV snapshots and auto-sweeps excess liquidity.
+ * @dev Only callable by the oracle role; uses epoch NAV snapshots.
  * @param requestIds Ordered list of deposit request IDs to process.
  */
 function processDeposits(uint256[] calldata requestIds) external override
```

onlyRole(ORACLE_ROLE) whenNotPaused {

3S-Vault-N02

Absence of request cancellation mechanism in **rHYPURR**

Id	3S-Vault-N02
Classification	None
Category	Suggestion
Status	Addressed in #16baabc.

Description

The **rHYPURR** contract implements an asynchronous vault where users submit deposit and redemption requests via **rHYPURR::requestDeposit**, **rHYPURR::requestDepositNative**, and **rHYPURR::requestRedemption**. Upon calling these functions, assets (WHYPE) or vault shares are transferred from the caller into the contract and a new request is enqueued with **RequestState.Pending**. Processing is performed later by a privileged **ORACLE_ROLE** account through **rHYPURR::processDeposits** and **rHYPURR::processRedemptions**, which follow strict FIFO ordering.

Once a request is created, the user has no way to cancel it and reclaim the transferred assets or shares. The **RequestState** enum only defines three values [**Uninitialized**, **Pending**, and **Processed**] and no function exists to transition a pending request back to a state that returns funds to the owner. This means that if a user submits a request and then changes his mind, he is forced to wait for processing with no opt-out path. Because queue processing depends entirely on the oracle and follows FIFO order, a user's funds can remain locked for an indeterminate period with no recourse.

Recommendation

Introduce a cancellation function that allows the original **owner** of a pending request to cancel it and recover the deposited assets or shares. Add a **Cancelled** value to the **RequestState** enum and ensure the queue advancement logic in **_advanceDepositQueue** and **_advanceRedemptionQueue** skips cancelled requests, which is already the case for non-**Pending** entries.