our shielding · Your smart contracts, our shielding · Your smart c

shieldify

Harmonix Finance

SECURITY REVIEW

Date: 9 May 2025

CONTENTS

1. About Shieldify

Positioned as the first hybrid Web3 Security company, Shieldify shakes things up with a unique subscription-based auditing model that entitles the customer to unlimited audits within its duration, as well as top-notch service quality thanks to a disruptive 6-layered security approach. The company works with very well-established researchers in the space and has secured multiple millions in TVL across protocols, also can audit codebases written in Solidity, Rust, Go, Vyper, Move and Cairo.

Learn more about us at shieldify.org.

2. Disclaimer

This security review does not guarantee bulletproof protection against a hack or exploit. Smart contracts are a novel technological feat with many known and unknown risks. The protocol, which this report is intended for, indemnifies Shieldify Security against any responsibility for any misbehavior, bugs, or exploits affecting the audited code during any part of the project's life cycle. It is also pivotal to acknowledge that modifications made to the audited code, including fixes for the issues described in this report, may introduce new problems and necessitate additional auditing.

3. About Harmonix Finance

Harmonix Finance is a DeFi platform that combines advanced hedge fund-grade strategies with native blockchain technology to optimize yield generation and liquidity efficiency. Designed for both retail and institutional investors, Harmonix offers tools to generate sustainable returns on assets like ETH, BTC, stablecoins, and DeFi positions such as staked ETH, restaked ETH, PT Pendle, and Hyperliquid tokens.

Harmonix stands apart by merging sophisticated derivative strategies, such as delta-neutral methods and out-of-the-money (OTM) options, with the transparency and accessibility of DeFi, ensuring users can earn more while mitigating risk.

4. Risk Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

4.1 Impact

- **High** – results in a significant risk for the protocol's overall well-being. Affects all or most users
- **Medium** – results in a non-critical risk for the protocol affects all or only a subset of users, but is still unacceptable
- **Low** – losses will be limited but bearable – and covers vectors similar to griefing attacks that can be easily repaired

4.2 Likelihood

- **High** – almost certain to happen and highly lucrative for execution by malicious actors
- **Medium** – still relatively likely, although only conditionally possible
- **Low** – requires a unique set of circumstances and poses non-lucrative cost-of-execution to rewards ratio for the actor

5. Security Review Summary

The security review lasted 5 days with a total of 80 hours dedicated by 2 researchers from the Shieldify team.

Overall, the code is well-written. The audit report identified eight critical and high-severity issues, six medium-severity issues, and two low-severity vulnerability. The vulnerabilities primarily stem from multiple deposit and withdrawal misconfigurations.

The Harmonix team has been highly responsive to the Shieldify research team's inquiries and promptly implemented all recommendations.

5.1 Protocol Summary

Project Name	**Harmonix Finance**
Repository	core-smart-contract
Type of Project	Yield Optimizer on Hyperliquid
Audit Timeline	5 days
Review Commit Hash	f02157aba919dcdd4a1133669361224108c5caef
Fixes Review Commit Hash	9fd411cbf63a6fdcbc255e6ab30ce41f26a73608

5.2 Scope

The following smart contracts were in the scope of the security review:

File	nSLOC
contracts/vaults/hyperliquid/BalanceContrac.sol	147
contracts/vaults/hyperliquid/FundContract.sol	510
contracts/vaults/hyperliquid/FundStorage.soll	57
Total	**714**

6. Findings Summary

The following number of issues have been identified, sorted by their severity:

- **Critical** and **High** issues: **8**
- **Medium** issues: **6**
- **Low** issues: **2**

ID	Title	Severity	Status
[C-01]	First Depositor Can Inflate Share Price to Steal Funds from Subsequent Depositors	Critical	Fixed
[C-02]	Users Can Exploit Rounding to Withdraw Excess Assets from the Fund Contract	Critical	Fixed
[C-03]	NAV Value Overwrites Last Fee Harvest Time	Critical	Fixed
[H-01]	Incorrect Withdrawal State Update Causes Partial Withdrawals to Fail	High	Fixed
[H-02]	Incorrect Fee Calculation Period Leads to Overcharging of Users	High	Fixed
[H-03]	Partial Withdrawers Suffer Double Fee Charging Due to State Mismanagement	High	Fixed
[H-04]	Faulty NAV Update Logic Creates Compounding Asset Depletion	High	Fixed
[H-05]	Admin Can Overcharge Users Due to Incorrect Fee Application in `FundContract`	High	Fixed
[M-01]	Admin Cannot Pause Vault Operations to Mitigate Protocol Risk	Medium	Fixed
[M-02]	Caller Can Permanently Lock Excess ETH in Contract	Medium	Fixed
[M-03]	Users Can Block Full Utilization of Vault Capacity	Medium	Fixed
[M-04]	Withdrawal Mechanism Fails When Share Price Declines	Medium	Fixed
[M-05]	Withdrawal Initiators Suffer Partial Loss When Share Price Increases	Medium	Fixed
[M-06]	Users Can Suffer Slippage Losses Due to Missing Slippage Protection in `FundContract`	Medium	Fixed
[L-01]	Batch Function Fails to Verify Total `msg.value`	Low	Fixed
[L-02]	Unused Storage Functions Increase Contract Bloat and Deployment Costs	Low	Fixed

7. Findings

[C-01] First Depositor Can Inflate Share Price to Steal Funds from Subsequent Depositors

Severity

Critical Risk

Description

The `FundContract` is susceptible to a share inflation attack due to its improper handling of share issuance during deposits and withdrawals. The attack arises because the contract overrides the ERC-4626Upgradeable `deposit()` function without implementing safeguards against inflation attacks.

An attacker can exploit this by:

- Depositing a minimal amount to receive shares.
- Withdrawing almost all shares, leaving 1 wei share.
- Donating a large amount of assets directly to the `balanceContract`, artificially inflating the NAV (Net Asset Value).
- When `updateNav()` triggers, which recalculates the share price based on the inflated NAV, which increases the share price

```
uint256 nav = IFundNavContract(fundNavContract).getFundNavValue() +
        ERC20(asset()).balanceOf(balanceContract) +
        withdrawPoolAmount -
        excludedTradingFee;

uint256 newPricePerShare = (nav * 10 ** decimals()) / totalSupply();
```

- When a legitimate user deposits, they receive zero/fewer shares than expected due to the inflated price.
- The attacker then withdraws their remaining shares, draining their and the user's amount of assets.
- Even though the contract inherits ERC4626-Upgradeable, the attack is still possible as the functions have been overridden.

Location of Affected Code

File: vaults/hyperliquid/fundContract.sol

```
function _updateNav() private {
// code
    uint256 nav = IFundNavContract(fundNavContract).getFundNavValue() +
                ERC20(asset()).balanceOf(balanceContract) +
                withdrawPoolAmount -
                excludedTradingFee;

            uint256 newPricePerShare = (nav * 10 ** decimals()) /
               totalSupply();
// code
}
```

File: vaults/hyperliquid/fundContract.sol

```
function deposit(
    uint256 _amount,
    address _receiver
) public override nonReentrant returns (uint256) {
    TransferHelper.safeTransferFrom(
        asset(),
        msg.sender,
        address(this),
        _amount
    );

    return _deposit(_amount, _receiver);
}
```

Impact

This attack has two implications: Implicit minimum Amount and funds lost due to rounding errors If an attacker is successful in making 1 share worth z assets and a user tries to mint shares using kz assets, then, - If k<1, then the user gets zero share and they lose all of their tokens to the attacker - If k>1, then users still get some shares, but they lose (k- floor(k)) z of assets, which get proportionally divided between existing shareholders (including the attacker) due to rounding errors. This means that, so that users do not lose value, they have to make sure that k is an integer.

Recommendation

Mint some shares (10**3) to the zero address after deploying the contract.

Team Response

Fixed.

[C-02] Users Can Exploit Rounding to Withdraw Excess Assets from the Fund Contract

Severity

Critical Risk

Description

The `withdraw()` function in `FundContract` converts the requested asset amount into shares using `convertToShares()`, which currently rounds down. However, the ERC4626 standard mandates that `convertToShares()` should round up during withdrawals to prevent users from withdrawing marginally more assets than their shares represent.

Since `withdraw()` uses `convertToShares(assets)` to determine how many shares to burn, rounding down allows users to receive slightly more assets per share than intended. Over multiple withdrawals, this discrepancy compounds, leading to an imbalance between the fund's total assets and its share supply.

Location of Affected Code

File: vaults/hyperliquid/fundContract.sol

```
function withdraw(
    uint256 assets,
    address receiver,
    address owner
) public override returns (uint256) {
// @audit round up should be done instead of round down
    uint256 shares = convertToShares(assets);
    return _withdraw(shares, receiver, owner);
}
```

Impact

If exploited repeatedly, this rounding error enables users to extract more value than their shares should permit, gradually draining the contract's assets. This results in financial losses for the protocol and disadvantages other shareholders by diluting the fund's NAV.

Recommendation

To fix this issue, `convertToShares()` should round up when used in `withdraw()`, aligning with the ERC4626 standard.

Team Response

Fixed.

[C-03] NAV Value Overwrites Last Fee Harvest Time

Severity

Critical Risk

Description

The `VaultStore` library defines a critical storage key incorrectly:

```
bytes32 public constant LAST_HARVEST_PERFORMANCE_FEE_TIME = keccak256(abi
    .encode("NAV"));
```

Instead of using a unique identifier for the last performance fee harvest time, it reuses the same key as NAV.

When `setVaultState()` is called (e.g., in `_updateVaultState()`), it stores the NAV value in `LAST_HARVEST_PERFORMANCE_FEE_TIME`. Consequently, `getLastHarvestPerformanceFeeTime()` returns the NAV value instead of the expected timestamp.

This causes `harvestPerformanceFee()` to always revert because:

```
require(block.timestamp >= lastHarvest + minHarvestInterval, "
    HARVEST_TOO_SOON");
```

Since `lastHarvest` is set to a large NAV value, the condition will never pass.

Location of Affected Code

File: lib/VaultStore.sol

```
// @audit this would return NAV value instead of
    LAST_HARVEST_PERFORMANCE_FEE_TIME value
bytes32 public constant LAST_HARVEST_PERFORMANCE_FEE_TIME = keccak256(abi
    .encode("NAV"));
```

```
function setVaultState(
    FundStorage fundStorage,
    bytes32 key,
    VaultState memory vaultState
) internal {
    fundStorage.setUint256(
        keccak256(abi.encode(key, PRICE_PER_SHARE)),
        vaultState.pricePerShare
    );
    fundStorage.setUint256(
        keccak256(abi.encode(key, WITHDRAW_POOL_AMOUNT)),
        vaultState.withdrawPoolAmount
    );
    fundStorage.setUint256(
        keccak256(abi.encode(key, LAST_HARVEST_MANAGEMENT_FEE_TIME)),
        vaultState.lastHarvestManagementFeeTime
    );
    fundStorage.setUint256(
        keccak256(abi.encode(key, LAST_HARVEST_PERFORMANCE_FEE_TIME)),
        vaultState.lastHarvestPerformanceFeeTime
    );
    fundStorage.setUint256(keccak256(abi.encode(key, NAV)), vaultState.
        nav);
}
```

Impact

Performance fees cannot be collected, leading to lost protocol revenue.

Recommendation

Fix the storage key to use the correct identifier:

```
bytes32 public constant LAST_HARVEST_PERFORMANCE_FEE_TIME = keccak256(abi
    .encode("LAST_HARVEST_PERFORMANCE_FEE_TIME"));
```

Team Response

Fixed.

[H-01] Incorrect Withdrawal State Update Causes Partial Withdrawals to Fail

Severity

High Risk

Description

The `FundContract` allows users to initiate withdrawals, which locks their shares and sets withdrawal data in storage. When the `acquireWithdrawalFunds()` function is called by the `balanceContract`, it marks the withdrawal as ready by setting `isAcquired = true` for the user. However, when a user partially withdraws their shares via the `withdraw()` function, the contract incorrectly resets `isAcquired` to false even if the user still has remaining shares to withdraw.

The issue arises in `_updateUserWithdrawal()`, which unconditionally sets `isAcquired = false` after a withdrawal, regardless of whether the user still has pending shares. Consequently, if a user attempts to withdraw their remaining shares, the check in `_canCompleteWithdraw()` will fail, preventing them from completing the withdrawal.

Location of Affected Code

File: vaults/hyperliquid/fundContract.sol

```
function _updateUserWithdrawal(
    address user,
    UserWithdraw.WithdrawData memory userWithdraw,
    uint256 _shares,
    uint256 withdrawAmount
) internal {
    userWithdraw.withdrawAmount -= withdrawAmount;
    userWithdraw.shares -= _shares;
// @audit partial withdrawal would fail
    userWithdraw.isAcquired = false;

    WithdrawStore.set(
        fundStorage,
        WithdrawStore.getWithdrawKey(address(this), user),
        userWithdraw
    );
}
```

File: vaults/hyperliquid/fundContract.sol

```
function _canCompleteWithdraw(address user) private view returns (bool) {
    UserWithdraw.WithdrawData memory userWithdraw = _getUserWithdraw(user
        );
    return userWithdraw.isAcquired;
}
```

Impact

Users who perform partial withdrawals will be unable to withdraw their remaining shares due to the incorrect state update. This constitutes a denial of service, as legitimate withdrawals are blocked.

Recommendation

The `_updateUserWithdrawal()` function should only set `isAcquired = false` if the user has fully withdrawn all their shares (i.e., when `userWithdraw.shares` equals zero). Otherwise, `isAcquired` should remain true to allow subsequent withdrawals.

Team Response

Fixed.

[H-02] Incorrect Fee Calculation Period Leads to Overcharging of Users

Severity

High Risk

Description

The `FundContract` initializes `lastHarvestManagementFeeTime` and `lastHarvestPerformanceFeeTime` to `block.timestamp` during `setupVaultSetting()`. This creates an issue when deposits occur significantly later, as the fee calculation in `harvestManagementFee()` and `harvestPerformanceFee()` uses the full time delta since contract initialization rather than the actual period during which funds were managed.

The problem manifests in `_calculateManagementFeeAmount()`, where the period variable is calculated as `timestamp - lastHarvestManagementFeeTime`. Since `lastHarvestManagementFeeTime` was set at contract initialization (potentially long before any deposits), this results in fees being calculated over an artificially extended period, leading to overcharging.

Location of Affected Code

File: vaults/hyperliquid/fundContract.sol

```
function _calculateManagementFeeAmount(
    uint256 timestamp,
    uint256 nav
) private view returns (uint256) {
    uint256 managementFeeRate = VaultStore.getManagementFeeRate(
        fundStorage,
        VaultStore.getVaultKey(address(this))
    );

    uint256 lastHarvestManagementFeeTime = VaultStore
        .getLastHarvestManagementFeeTime(
            fundStorage,
            VaultStore.getVaultKey(address(this))
        );
    uint256 perSecondRate = (managementFeeRate * 1e12) / (365 * 86400) +
       1; // +1 mean round up second rate
    uint256 period = timestamp - lastHarvestManagementFeeTime;
    return (nav * perSecondRate * period) / 1e14;
}
```

File: vaults/hyperliquid/fundContract.sol

```
function setupVaultSetting(
    uint256 _minimumSupply,
    uint256 _capacity,
    uint256 _performanceFeeRate,
    uint256 _managementFeeRate,
    uint8 _tradingFeeRate,
    address _feeReceiver,
    uint256 _networkCost
) external nonReentrant {
    _auth(Role.ADMIN);
    bytes32 vaultKey = VaultStore.getVaultKey(address(this));
    VaultStore.VaultSetting memory vaultSetting = VaultStore.VaultSetting
       (
        _minimumSupply,
        _capacity,
        _performanceFeeRate,
        _managementFeeRate,
        _tradingFeeRate,
        _feeReceiver,
        false,
        _networkCost
    );
    VaultStore.setVaultSetting(fundStorage, vaultKey, vaultSetting);
```

```
        VaultStore.VaultState memory vaultState = VaultStore.VaultState(
            10 ** decimals(),
            0,
            0,
            // @audit Fee would be charged more
            block.timestamp,
            block.timestamp,
            0
        );

        VaultStore.setVaultState(fundStorage, vaultKey, vaultState);
    }
```

Impact

Users will be charged management and performance fees for periods when their funds were not deposited in the contract. This constitutes financial loss for users, as they pay fees disproportionate to the actual service period. The overcharging could significantly reduce user returns, especially if there's a long gap between contract deployment and first deposits.

Recommendation

The fee calculation should only consider periods when funds were actually deposited. This can be achieved by initializing harvest timestamps on the first deposit rather than the contract setup.

Team Response

Fixed.

[H-03] Partial Withdrawers Suffer Double Fee Charging Due to State Mismanagement

Severity

High Risk

Description

The vulnerability manifests in the interaction between `initiateWithdrawal()` and subsequent partial withdrawals via `redeem()`. When a user initiates withdrawal, the contract stores cumulative fee amounts in `userWithdraw.managementFee`. During partial withdrawals, while `_shares` and `userWithdraw.withdrawAmount` are properly reduced in `_updateUserWithdrawal()`, the critical `userWithdraw.managementFee` remains unchanged.

This creates an accounting mismatch where each partial withdrawal recalculates fees using the original full fee amount
(`sharesManagementFee = (_shares * userWithdraw.managementFee) / userWithdraw.shares`), despite having already charged portions of these fees in previous withdrawals. The unmodified **managementFee** state causes the protocol to repeatedly deduct fees from the same original allocation.

Location of Affected Code

File: vaults/hyperliquid/fundContract.sol

```
function _updateUserWithdrawal(
    address user,
    UserWithdraw.WithdrawData memory userWithdraw,
    uint256 _shares,
    uint256 withdrawAmount
) internal {
    userWithdraw.withdrawAmount -= withdrawAmount;
    userWithdraw.shares -= _shares;
    userWithdraw.isAcquired = false;

    WithdrawStore.set(
        fundStorage,
        WithdrawStore.getWithdrawKey(address(this), user),
        userWithdraw
    );
}
```

Impact

Users performing multiple partial withdrawals suffer progressive financial losses as fees are compounded with each transaction.

Recommendation

The remediation requires modifying the state management logic to properly account for partial withdrawals and ensure proportional fee deductions. The core fix involves updating `_updateUserWithdrawal()` to adjust all stored values – including shares, withdrawal amounts, and fees – based on the proportion of shares being redeemed.

Team Response

Fixed.

[H-04] Faulty NAV Update Logic Creates Compounding Asset Depletion

Severity

High Risk

Description

The vulnerability manifests in the interaction between withdrawal initiation and execution. During `initiateWithdrawal()`, the calculated `withdrawAmount` already includes the management fee (as evidenced by managementFee being calculated from withdrawAmount but not subtracted from it). However, in `_withdraw()`, the protocol again subtracts both the withdrawal amount and fees

from the NAV through `_updateVaultState(vaultState, sharesWithdrawAmount, totalManagementFee)`
.

The internal `_updateVaultState()` function incorrectly treats the withdrawal amount and fees as separate deductions `(nav -= (withdrawAmount + feeAmount))`, despite the withdrawal amount already incorporating the fee component. This double-counting progressively erodes the fund's NAV with each withdrawal.

Location of Affected Code

File: vaults/hyperliquid/fundContract.sol

```
function initiateWithdrawal(uint256 _shares) external nonReentrant {
    require(balanceOf(msg.sender) >= _shares, "INVALID_SHARES");
    UserWithdraw.WithdrawData memory userWithdraw = _getUserWithdraw(
        msg.sender
    );
// @audit create problem if second time he comes for withdrawal at that
   time, due to rounding the exact number of shares can't come, due to
   that he can not withdraw a second time
    require(userWithdraw.shares == 0, "INVALID_WD_STATE");
// @audit confirm should round up here
    uint256 withdrawAmount = ShareMath.sharesToAsset(
        _shares,
        _getPricePerShare(),
        decimals()
    );

    uint256 managementFee = _calculateManagementFeeAmount(
        block.timestamp,
        withdrawAmount
    );

// @audit fees not taken on withdrawAmount , it is taken differently so
   would create a prob
    _createUserWithdraw(
        msg.sender,
        userWithdraw,
        _shares,
        withdrawAmount,
        managementFee
    );

    emit WithdrawalInitiated(msg.sender, withdrawAmount, _shares);
}
```

File: vaults/hyperliquid/fundContract.sol

```
function _updateVaultState(
    VaultStore.VaultState memory vaultState,
    uint256 withdrawAmount,
    uint256 feeAmount
) internal {
    vaultState.withdrawPoolAmount -= withdrawAmount;
    vaultState.nav -= (withdrawAmount + feeAmount);

    VaultStore.setVaultState(
        fundStorage,
        VaultStore.getVaultKey(address(this)),
        vaultState
    );
}
```

Impact

The protocol's NAV will decrease at an accelerated rate as each withdrawal improperly deducts fees twice from the total assets. This creates a compounding effect where user withdrawals systematically remove more value from the fund than intended

Recommendation

The `_updateVaultState()` function must be modified to recognize that fees are already included in the withdrawal amount. Instead of deducting both, it should only deduct the net withdrawal amount after the fee payment.

Team Response

Fixed.

[H-05] Admin Can Overcharge Users Due to Incorrect Fee Application in `FundContract`

Severity

High Risk

Description

The `FundContract` incorrectly applies `management fees` instead of `trading fees` during withdrawals, leading to users being overcharged. The issue arises in the `_withdraw()` function, where the sharesTradingFee is incorrectly calculated as `(_shares * userWithdraw.managementFee) / userWithdra`, instead of using the trading fee stored in the user's withdrawal data.

Additionally, the trading fee is not properly set during withdrawal initiation (`nitiateWithdrawal()`). While the management fee is calculated and stored in the `UserWithdraw.WithdrawData` struct, the trading fee remains unset, causing the contract to default to using the management fee again during withdrawals.

Location of Affected Code

File: vaults/hyperliquid/fundContract.sol

```
uint256 sharesTradingFee = (_shares * userWithdraw.managementFee) /
    userWithdraw.shares;
```

Impact

Since the trading fee (0.035%) is significantly lower than the management fee (1%), users are charged substantially higher fees than intended. This results in financial losses for users withdrawing funds, as they pay fees at a rate nearly 30 times higher than expected.

Recommendation

To mitigate this issue, consider applying the following changes: - Store the correct trading fee during withdrawal initiation by modifying `initiateWithdrawal()` to include the trading fee in the `UserWithdraw.WithdrawData` struct. - Use the stored trading fee instead of the management fee in the `_withdraw()` function when calculating `sharesTradingFee`.

Team Response

Fixed.

[M-01] Admin Cannot Pause Vault Operations to Mitigate Protocol Risk

Severity

Medium Risk

Description

The `VaultSetting` struct within the `Vault` contract includes an `isPaused` boolean field intended to allow administrators to pause core vault operations such as deposits and withdrawals during abnormal or high-risk scenarios. The `setupVaultSetting()` function is responsible for initializing and updating vault parameters, including minimumSupply, capacity, and various fee configurations. However, this function hardcodes the `isPaused` field to false, and there exists an administrative function to set **`isPaused`** to true.

This design omission results in a critical gap in operational controls. Although the contract appears to support a paused state, in practice, there is no mechanism for administrators to activate it. As a result, the pause functionality is entirely unusable.

Location of Affected Code

File: vaults/hyperliquid/fundContract.sol

```
function setupVaultSetting(
    uint256 _minimumSupply,
    uint256 _capacity,
    uint256 _performanceFeeRate,
    uint256 _managementFeeRate,
    uint8 _tradingFeeRate,
    address _feeReceiver,
    uint256 _networkCost
) external nonReentrant {
      _auth(Role.ADMIN);
      bytes32 vaultKey = VaultStore.getVaultKey(address(this));
      VaultStore.VaultSetting memory vaultSetting = VaultStore.
         VaultSetting(
          _minimumSupply,
          _capacity,
          _performanceFeeRate,
          _managementFeeRate,
          _tradingFeeRate,
          _feeReceiver,
          false,
          _networkCost
      );
      VaultStore.setVaultSetting(fundStorage, vaultKey, vaultSetting);
// code
}
```

Impact

The inability to pause vault operations undermines the protocol's resilience to emergency situations. In the event of an exploit or a required upgrade, administrators would be unable to halt deposits and withdrawals, potentially exposing user funds to loss or exploitation.

Recommendation

Introduce a dedicated administrative function, such as `pauseVault()` and `unpauseVault()`, that allows authorized roles (e.g., `Role.ADMIN`) to explicitly update the `isPaused` state for the vault.

Team Response

Fixed.

[M-02] Caller Can Permanently Lock Excess ETH in Contract

Severity

Medium Risk

Description

In the `depositNative()` function, a user is allowed to send more ETH (`msg.value`) than the specified `_amount` they intend to deposit. The function checks that `msg.value >= _amount` but does

not enforce strict equality, nor does it refund any excess ETH. The entire `msg.value` is passed to the `IWETH(asset()).deposit{value: msg.value}()` call, which wraps the full amount into WETH. However, only `_amount` of WETH is then deposited via the internal `_deposit()` function, with the rest of the wrapped ETH remaining in the contract's balance, potentially without a corresponding accounting mechanism or method to recover or refund the excess.

This oversight can result in ETH being unintentionally locked within the contract. Users may accidentally overpay and have no means to retrieve the excess, leading to a potential financial loss.

Location of Affected Code

File: vaults/hyperliquid/fundContract.sol

```
function depositNative(
      uint256 _amount,
      address _receiver
) external payable nonReentrant returns (uint256) {
// @audit confirm excess eth not transferred back
    require(msg.value >= _amount, "INVALID_AMOUNT");

    IWETH(asset()).deposit{value: msg.value}();

    return _deposit(_amount, _receiver);
}
```

Impact

Users who send more ETH than the `_amount` parameter in `depositNative()` will have the surplus ETH irretrievably locked in the contract. This leads to unnecessary user losses and undermines trust in the protocol's handling of native tokens.

Recommendation

To resolve this issue, the function should enforce exact equality between `msg.value` and `_amount` by replacing the conditional `require(msg.value >= _amount, "INVALID_AMOUNT");` with `require(msg.value == _amount, "INVALID_AMOUNT");`. This ensures that users do not accidentally send more ETH than intended. Alternatively, if partial deposits are acceptable, the contract should calculate and refund any surplus ETH after the deposit operation is complete.

Team Response

Fixed.

[M-03] Users Can Block Full Utilization of Vault Capacity

Severity

Medium Risk

Description

The vulnerability is located in the `_deposit()` function, which enforces two constraints: a lower-bound check on the deposit amount `(require(vaultSetting.minimumSupply <= _amount, "INVALID_AMOUNT");)` and an upper-bound check to ensure that the vault's capacity is not exceeded **(require(VaultStore.getNav(fundStorage, vaultKey) + _amount <= vaultSetting.capacity, "INVALID_AMOUNT_CAPACITY");)**. While these checks are valid in isolation, their interaction introduces a logic flaw when the vault approaches its capacity limit.

If the remaining capacity in the vault falls below the configured `minimumSupply`, it becomes impossible for any user to deposit additional funds, even if the remaining space could accommodate a smaller deposit. This occurs because any deposit smaller than `minimumSupply` is explicitly rejected, while any deposit equal to or greater than `minimumSupply` will breach the vault's capacity limit and also be rejected.

For example, consider a vault with a `capacity` of 1,000 units and a `minimumSupply` of 100 units. Suppose the current NAV of the vault is 950 units. This leaves only 50 units of available space. At this point, no further deposit is possible: a deposit of 50 will be rejected for being less than the minimum supply, and a deposit of 100 or more will be rejected for exceeding the vault's capacity. As a result, the remaining 50 units of capacity become permanently unusable.

Location of Affected Code

File: vaults/hyperliquid/fundContract.sol

```
function _deposit( uint256 _amount, address _receiver ) private returns (
    uint256) {
// code
      require(vaultSetting.minimumSupply <= _amount, "INVALID_AMOUNT");

      uint256 tradingFee = (_amount * vaultSetting.tradingFeeRate) / 1e5;

      require(VaultStore.getNav(fundStorage, vaultKey) + _amount <=
            vaultSetting.capacity,
          "INVALID_AMOUNT_CAPACITY"
      );
// code
}
```

Impact

This flaw allows users to inadvertently—or maliciously—trigger a state in which part of the vault's capacity is permanently locked. It prevents full capital deployment and introduces inefficiencies in fund management.

Recommendation

To resolve this issue, the logic in `_deposit()` should account for scenarios in which the remaining vault capacity is below the minimum supply. A safe and effective fix involves conditionally adjusting the minimum deposit requirement based on the vault's remaining capacity.

Team Response

Fixed.

[M-04] Withdrawal Mechanism Fails When Share Price Declines

Severity

Medium Risk

Description

The vulnerability manifests in the withdrawal flow between `initiateWithdrawal()` and `withdraw()`. When a user initiates withdrawal, the contract records the share amount and equivalent asset value at that time. However, if the share price decreases before execution, `withdraw()` recalculates the share amount using the new lower price via `convertToShares()`, resulting in more shares than originally recorded.

The critical check `require(userWithdraw.shares >= _shares, "INVALID_SHARES")` then fails because the recalculated share amount exceeds the initially recorded value. This creates a deadlock where users cannot withdraw funds despite having initiated the process, as the system compares share amounts from different price epochs without adjustment.

Location of Affected Code

File: vaults/hyperliquid/fundContract.sol

```
function initiateWithdrawal(uint256 _shares) external nonReentrant {
    require(balanceOf(msg.sender) >= _shares, "INVALID_SHARES");
    UserWithdraw.WithdrawData memory userWithdraw = _getUserWithdraw(
        msg.sender
    );
    require(userWithdraw.shares == 0, "INVALID_WD_STATE");
    uint256 withdrawAmount = ShareMath.sharesToAsset(
        _shares,
        _getPricePerShare(),
        decimals()
    );

    uint256 managementFee = _calculateManagementFeeAmount(
        block.timestamp,
        withdrawAmount
    );
```

```
        _createUserWithdraw(
            msg.sender,
            userWithdraw,
            _shares,
            withdrawAmount,
            managementFee
        );

        emit WithdrawalInitiated(msg.sender, withdrawAmount, _shares);
    }


    function withdraw(
        uint256 assets,
        address receiver,
        address owner
    ) public override returns (uint256) {
        uint256 shares = convertToShares(assets);
        return _withdraw(shares, receiver, owner);
    }
```

File: vaults/hyperliquid/fundContract.sol

```
    function _withdraw(
        uint256 _shares,
        address _receiver,
        address _owner
    ) private returns (uint256) {
        require(msg.sender == _owner, "INVALID_OWNER");

        UserWithdraw.WithdrawData memory userWithdraw = _getUserWithdraw(
            msg.sender
        );
    // @audit confirm this would fail if share price decreases
        require(userWithdraw.shares >= _shares, "INVALID_SHARES");

    // code
    }
```

Impact

Affected users become unable to complete withdrawals when share prices decline, effectively freezing their funds until prices recover. This violates the expected withdrawal guarantee and damages user trust.

Recommendation

The solution requires decoupling the share validation from price fluctuations. Instead of recalculating shares during withdrawal, the contract should store the original share amount from `initiateWithdrawal()` and use it directly in `withdraw()` without recalculation.

Team Response

Fixed.

[M-05] Withdrawal Initiators Suffer Partial Loss When Share Price Increases

Severity

Medium Risk

Description

The vulnerability manifests in the interaction between `initiateWithdrawal()` and `withdraw()`. When share prices increase after initiation but before execution, the proportional calculation in `_withdraw()` divides the originally recorded withdrawal amount by the new, higher share price. This results in users receiving substantially fewer assets than their initial entitlement.

The critical flaw occurs in the shares-to-assets conversion, where `sharesWithdrawAmount = (_shares * userWithdraw.withdrawAmount) / userWithdraw.shares` fails to account for NAV increases. Since userWithdraw.withdrawAmount remains fixed at initiation-time values while share prices appreciate, the equation yields diminished asset payouts despite the fund's increased value.

Location of Affected Code

File: vaults/hyperliquid/fundContract.sol

```
function initiateWithdrawal(uint256 _shares) external nonReentrant {
    require(balanceOf(msg.sender) >= _shares, "INVALID_SHARES");
    UserWithdraw.WithdrawData memory userWithdraw = _getUserWithdraw(
        msg.sender
    );
    require(userWithdraw.shares == 0, "INVALID_WD_STATE");
    uint256 withdrawAmount = ShareMath.sharesToAsset(
        _shares,
        _getPricePerShare(),
        decimals()
    );

    uint256 managementFee = _calculateManagementFeeAmount(
        block.timestamp,
        withdrawAmount
    );
    _createUserWithdraw(
        msg.sender,
        userWithdraw,
        _shares,
        withdrawAmount,
        managementFee
    );

    emit WithdrawalInitiated(msg.sender, withdrawAmount, _shares);
}

function withdraw(
    uint256 assets,
    address receiver,
    address owner
) public override returns (uint256) {
    uint256 shares = convertToShares(assets);
    return _withdraw(shares, receiver, owner);
}
```

File: vaults/hyperliquid/fundContract.sol

```
function _withdraw(
    uint256 _shares,
    address _receiver,
    address _owner
) private returns (uint256) {
    require(msg.sender == _owner, "INVALID_OWNER");

    UserWithdraw.WithdrawData memory userWithdraw = _getUserWithdraw(
        msg.sender
    );
// @audit confirm this would fail if share price decreases
    require(userWithdraw.shares >= _shares, "INVALID_SHARES");
    .
    .
    uint256 sharesWithdrawAmount = (_shares * userWithdraw.withdrawAmount
       ) /
        userWithdraw.shares;

// code
}
```

Impact

Users suffer direct financial losses when share prices rise, as their withdrawals become locked to outdated valuations due to only the partially withdrawn amount, which is set during `initiateWithdrawal()`, would be claimed, and the other remaining amount would be lost to the user.

Recommendation

The solution requires decoupling the share validation from price fluctuations. Instead of recalculating shares during withdrawal, the contract should store the original share amount from `initiateWithdrawal()` and use it directly in `withdraw()` without recalculation.

Team Response

Fixed.

[M-06] Users Can Suffer Slippage Losses Due to Missing Slippage Protection in `FundContract`

Severity

Medium Risk

Description

The `FundContract` lacks slippage protection mechanisms in its deposit and withdrawal functions (`deposit()`, `depositNative()`, `withdraw()`, and `redeem()`), which exposes users to potential financial losses due to unfavourable exchange rates between assets and shares.

The vulnerability arises because the contract implements the ERC-4626 standard but does not provide a way for users to specify minimum acceptable amounts of shares (for deposits) or assets (for withdrawals). While the standard permits direct EOA (Externally Owned Account) interaction, it strongly recommends implementing slippage controls if such access is supported. Currently, users calling these functions directly may receive fewer shares or assets than expected due to price fluctuations between transaction submission and execution.

Location of Affected Code

File: vaults/hyperliquid/fundContract.sol

```
function deposit(
    uint256 _amount,
    address _receiver
) public override nonReentrant returns (uint256) {
    TransferHelper.safeTransferFrom(
        asset(),
        msg.sender,
        address(this),
        _amount
    );

    return _deposit(_amount, _receiver);
}
```

Impact

Users depositing or withdrawing funds may experience unexpected losses if the exchange rate between assets and shares changes unfavorably before their transactions are processed.

Recommendation

To mitigate this issue, the contract should implement slippage protection by:

- Adding `minSharesOut` and `minAssetsOut` Parameters – Modify `deposit()` / `withdrawal()` functions to allow users to specify minimum acceptable amounts of shares (for deposits) or assets (for withdrawals). Revert if the actual amount received is below this threshold.

Team Response

Fixed.

[L-01] Batch Function Fails to Verify Total `msg.value`

Severity

Low Risk

Description

The `executeBatchActions()` and `executeAction()` functions process transactions with specified Ether values (`_values[]`) but do not validate whether the total `msg.value` matches the sum

of all `_values[]` / `_value`. If `msg.value` is greater than the sum of `_values[]` / `_value`, the excess Ether remains stuck in the contract with no way to recover it. If `msg.value` is less than the sum of `_values[]`, the transaction would revert.

Location of Affected Code

File: vaults/hyperliquid/balanceContract.sol

```
function executeAction(
    address _target,
    uint256 _value,
    bytes calldata _data
) external payable nonReentrant returns (bytes memory) {
    _auth(Role.OPERATOR);

    require(_target != address(0), "INVALID_TARGET");
    require(isActionAllowed(_target, _data), "FUNCTION_NOT_WHITELISTED");

// Execute the call
// @audit not checked if msg.value is equal to _value
    (bool success, bytes memory result) = _target.call{value: _value}(
        _data
    );
    require(success, "Transaction failed");

    return result;
}
```

File: vaults/hyperliquid/balanceContract.sol

```
function executeBatchActions(
    address[] calldata _targets,
    uint256[] calldata _values,
    bytes[] calldata _data
) external payable nonReentrant returns (bool[] memory) {
    _auth(Role.CONTROLLER);

    require(
        _targets.length == _values.length && _values.length == _data.
           length,
        "MISMATCHED_ARRAYS"
    );
```

```
    bool[] memory results = new bool[](_targets.length);
    for (uint256 i = 0; i < _targets.length; i++) {
        require(
            isActionAllowed(_targets[i], _data[i]),
            string(
                abi.encodePacked(
                    "FUNCTION_NOT_WHITELISTED_AT_INDEX_",
                    Strings.toString(i)
                )
            )
        );
        (bool success, ) = _targets[i].call{value: _values[i]}(_data[i]);

        require(success, "Transaction failed");

        results[i] = success;
    }

    return results;
}
```

Impact

If more ether has been passed, it would permanently lock excess Ether inside the contract, and if less ether has been passed, the transaction would revert.

Recommendation

Add a check that `msg.value` is equal to the summation of `_values[]` / `_value`.

Team Response

Fixed.

[L-02] Unused Storage Functions Increase Contract Bloat and Deployment Costs

Severity

Low Risk

Description

The `FundStorage` contract contains multiple unused functions that are never called anywhere in the protocol:

`String Operations (getString, setString, removeString)`;

`Bytes32 Array Operations (getBytes32Array, setBytes32Array, removeBytes32Array)`;

`Bool Array Operations (getBoolArray, setBoolArray, removeBoolArray)`;

`String Array Operations (getStringArray, setStringArray, removeStringArray)`;

These functions serve no functional purpose in the contract.

Location of Affected Code

File: vaults/hyperliquid/fundStorage.sol

Impact

Wasted gas during deployment and contract interactions.

Recommendation

Remove all unused functions if they are of no use.

Team Response

Fixed.

our shielding · Your smart contracts, our shielding · Your smart c

shieldify

Thank you!