Pashov Audit Group

Elytra Security Review

July 27th 2025 - August 3rd 2025

Contents

1. About Pashov Audit Group 3
2. Disclaimer 3
3. Risk Classification 3
4. About Elytra 4
5. Executive Summary 4
7. Findings 5
Medium findings 7
[M-01] Inconsistent validation between asset transfer functions 7
[M-02] Price change limit can cause system DoS 8
[M-03] Elytra share price may be inflatten to block the system 9
Low findings 11
[L-01] Inconsistent timestamps cause incorrect staleness checks and info 11
[L-02] Multiple rounding down issue for total value 12
[L-03] Incorrect yield tracking in case of amount > tracked allocation 12
[L-04] Incorrect gap slot amount due to incorrect variable counting 12
[L-05] Asymmetric yield treatment for queued withdrawals 12
[L-06] Inconsistent withdrawal control in `DepositPool` vs `UnstakingVault` 13
[L-07] `getElyAssetAmountToMint()` uses outdated `elyAsset` prices, causing issues 14
[L-08] No cap on protocol fee setting creates DoS risk 14
[L-09] Missing storage gaps in upgradeable oracle contracts 15
[L-10] Inconsistency between the comment and implementation 16
[L-11] Static allocation vs dynamic strategy causes DoS in loss reporting 16
[L-12] `kHYPE` on-chain price validation allows frontrun profit 17
[L-13] Request withdrawals may revert due to max capital efficiency 18
[L-14] `assetsAllocatedToStrategies` will be incorrect after loss scenario 19
[L-15] User assets may be stolen 19
[L-16] Users can frontrun the possible slash in strategy to avoid the loss 20
[L-17] Delayed slashing reporting enables timing-based loss avoidance 21
[L-18] Complete withdrawal fee bypass via unstaking vault 22
[L-19] Improper decimal process when we calculate the totalValueInProtocol 23

1. About Pashov Audit Group

Pashov Audit Group consists of 40+ freelance security researchers, who are well proven in the space - most have earned over $100k in public contest rewards, are multi-time champions or have truly excelled in audits with us. We only work with proven and motivated talent.

With over 300 security audits completed — uncovering and helping patch thousands of vulnerabilities — the group strives to create the absolute very best audit journey possible. While 100% security is never possible to guarantee, we do guarantee you our team's best efforts for your project.

Check out our previous work here or reach out on Twitter @pashovkrum.

2. Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

3. Risk Classification

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

Impact

- **High** - leads to a significant material loss of assets in the protocol or significantly harms a group of users
- **Medium** - leads to a moderate material loss of assets in the protocol or moderately harms a group of users
- **Low** - leads to a minor material loss of assets in the protocol or harms a small group of users

Likelihood

- **High** - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost
- **Medium** - only a conditionally incentivized attack vector, but still relatively likely
- **Low** - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive

4. About Elytra

Elytra is a Liquid Restaking Protocol for HyperEVM that enables users to deposit HYPE or USDC and receive elyHYPE (rsHYPE) or elyUSD tokens, respectively, representing their stake in the system. It features a modular architecture with integrated strategy management, price oracles, and a secure, role-based control system to facilitate asset restaking and withdrawals.

5. Executive Summary

A time-boxed security review of the **ElytraProtocol/contracts** repository was done by Pashov Audit Group, during which Pashov Audit Group engaged to review **Elytra**. A total of **22** issues were uncovered.

Protocol Summary	
Project Name	Elytra
Protocol Type	Liquid Restaking
Timeline	July 27th 2025 - August 3rd 2025

Review commit hash:
- 2bbfd0117c13122111251135b31bc6764fb25bc2 (ElytraProtocol/contracts)

Fixes review commit hash:
- 008f19e65ebd5172f3e822de0bd1263c61dec01a (ElytraProtocol/contracts)

Scope

`ElytraConfigV1.sol` `ElytraDepositPoolV1.sol` `ElytraOracleV1.sol` `ElytraUnstakingVaultV1.sol` `elyHYPE.sol` `elyUSD.sol` `ElyAsset.sol` `WHYPEPriceOracle.sol` `ElytraConfigRoleChecker.sol` `KHYPEPriceOracle.sol`

6. Findings

Findings count

Severity	Amount
Medium	3
Low	19
Total findings	**22**

Summary of findings

ID	Title	Severity	Status
[M-01]	Inconsistent validation between asset transfer functions	Medium	Resolved
[M-02]	Price change limit can cause system DoS	Medium	Resolved
[M-03]	Elytra share price may be inflatten to block the system	Medium	Resolved
[L-01]	Inconsistent timestamps cause incorrect staleness checks and info	Low	Resolved
[L-02]	Multiple rounding down issue for total value	Low	Resolved
[L-03]	Incorrect yield tracking in case of amount > tracked allocation	Low	Resolved
[L-04]	Incorrect gap slot amount due to incorrect variable counting	Low	Resolved
[L-05]	Asymmetric yield treatment for queued withdrawals	Low	Resolved
[L-06]	Inconsistent withdrawal control in `DepositPool` vs `UnstakingVault`	Low	Resolved
[L-07]	`getElyAssetAmountToMint()` uses outdated `elyAsset` prices, causing issues	Low	Resolved
[L-08]	No cap on protocol fee setting creates DoS risk	Low	Resolved
[L-09]	Missing storage gaps in upgradeable oracle contracts	Low	Resolved

ID	Title	Severity	Status
[L-10]	Inconsistency between the comment and implementation	Low	**Resolved**
[L-11]	Static allocation vs dynamic strategy causes DoS in loss reporting	Low	**Resolved**
[L-12]	`kHYPE` on-chain price validation allows frontrun profit	Low	Acknowledged
[L-13]	Request withdrawals may revert due to max capital efficiency	Low	**Resolved**
[L-14]	`assetsAllocatedToStrategies` will be incorrect after loss scenario	Low	**Resolved**
[L-15]	User assets may be stolen	Low	Acknowledged
[L-16]	Users can frontrun the possible slash in strategy to avoid the loss	Low	**Resolved**
[L-17]	Delayed slashing reporting enables timing-based loss avoidance	Low	Acknowledged
[L-18]	Complete withdrawal fee bypass via unstaking vault	Low	**Resolved**
[L-19]	Improper decimal process when we calculate the totalValueInProtocol	Low	**Resolved**

Medium findings

[M-01] Inconsistent validation between asset transfer functions

Severity

Impact: Medium

Likelihood: Medium

Description

The `ElytraDepositPoolV1` has inconsistent validation logic between `allocateToStrategy()` and `transferAssetToUnstakingVault()` when checking tracked deposit balances. This inconsistency allows attackers to bypass donation attack protections by using the lenient transfer path instead of the strict allocation path.

Let's check `ElytraDepositPoolV1.sol` code where **inconsistent validation logic** is present.

```
// allocateToStrategy() - STRICT validation
function allocateToStrategy(address asset, uint256 amount) external {
    if (totalAssetDepositsTracked[asset] < amount) revert
InsufficientTrackedDepositsForAllocation();
    // in here it bLOCKS operation if insufficient tracked deposits @audit-poc
}

// transferAssetToUnstakingVault() - LENIENT validation
function transferAssetToUnstakingVault(address asset, uint256 amount) external {
    if (amount <= totalAssetDepositsTracked[asset]) {
        totalAssetDepositsTracked[asset] -= amount;
    } else {
        totalAssetDepositsTracked[asset] = 0; // ❌ Just sets to 0 and continues @audit-poc
    }
    // here it ALLOWS operation regardless of tracked deposit amount
}
```

Let's take an example **attack scenario** 1. Attacker donates tokens directly to the contract. 2. `allocateToStrategy()` correctly blocks using donated tokens (cuz of the strict check). 3. `transferAssetToUnstakingVault()` allows using donated tokens (lenient check). 4. Pool's `totalAssetDepositsTracked` gets corrupted (set to 0 instead of proper tracking).

Now let's take a real life **example:**

```
Initial State: 1000 USDC tracked deposits, 1000 USDC actual balance
Attacker donates: 500 USDC directly to contract
New State: 1000 USDC tracked, 1500 USDC actual balance

allocateToStrategy(USDC, 1200) → REVERTS ✅ (1000 < 1200)
transferAssetToUnstakingVault(USDC, 1200) → SUCCEEDS ❌ (sets tracking to 0)
```

This enables donation attacks to corrupt deposit accounting by bypassing strict validation, allowing use of non-user deposits and breaking the L-06 fix intended to prevent exactly this scenario.

Recommendation

Make validation consistent between both functions by using the same strict check.

```
function transferAssetToUnstakingVault(address asset, uint256 amount) external {
    // Add the same validation as allocateToStrategy
    if (totalAssetDepositsTracked[asset] < amount) {
        revert InsufficientTrackedDepositsForAllocation();
    }

    // Standard decrement (no special case handling)
    totalAssetDepositsTracked[asset] -= amount;

    // Rest of function...
}
```

[M-02] Price change limit can cause system DoS

Severity

Impact: High

Likelihood: Low

Description

In Elytra, we will have one default price percentage limit, 10%. If the difference between the current price and updated price exceeds the price limit, deposit/withdraw will be blocked.

If there is something wrong with one LST, which causes the share's price drops, e.g 10%. This will block deposit/withdraw. This is one expected behavior.

As the admin, we wish to increase the price percentage or remove the limit to allow users withdraw their assets. The problem here is that in the function `setPricePercentageLimit`, we will try to update the price. This transaction will be reverted again. The admin will lose the ability to recover the system.

```
    function setPricePercentageLimit(uint256 _limit) external override onlyElytraAdmin {
@>      this.updateElyAssetPrice();
        pricePercentageLimit = _limit;
        emit PricePercentageLimitUpdated(_limit);
    }
```

Recommendations

It's not necessary to update `updateElyAssetPrice`.

[M-03] Elytra share price may be inflatten to block the system

Severity

Impact: High

Likelihood: Low

Description

In Elytra, total asset tvl is calculated based on the formula: `totalAssetDepositsTracked + unstakingVault + strategyBalance - reserveBalance`.

Users can choose withdrawAssets or requestWithdrawal to withdraw their assets. When we withdraw assets, we will check whether we have enough balance in ElyTra pool, and we will update the variable `totalAssetDepositsTracked`.

In a normal scenario, the pool's balanceOf should equal `totalAssetDepositsTracked` when we deposit/withdraw/transferUnstaking, we will update `totalAssetDepositsTracked` timely.

The problem here is that if users donate some assets into the pool, users can withdraw assets when there is not enough balance in the pool. This case can still work, and we will update the variable `totalAssetDepositsTracked` to `0`. However, this will impact share's price.

For example: 1. Alice, as the first depositor, deposits 1000 HYPE. 2. Admin allocates 1000 HYPE into the related strategy. Now `totalAssetDepositsTracked` is 0. 3. Alice donates 999 HYPE into the pool, and withdraws 999 shares. `totalAssetDepositsTracked` will keep `0`, and total share's amount will be decreased to 1. This will increase share's price. 4. If we set the `pricePercentageLimit`, users may fail to deposit/withdraw because of this price limit check. Another impact here is that when we increase share's price a lot, the actual asset for each rounding down/up will become larger. This will have some bad experiences for investors.

```
    function withdrawAsset(
        address asset,
        uint256 elyAssetAmount,
        address receiver,
        uint256 minUnderlyingAssetExpected
    )
    {
        uint256 poolBalance = IERC20(asset).balanceOf(address(this));
        if (poolBalance < assetAmountBeforeFee) {
            revert InsufficientPoolBalance(assetAmountBeforeFee, poolBalance);
        }
        ...
@>        if (assetAmountBeforeFee <= totalAssetDepositsTracked[asset]) {
            totalAssetDepositsTracked[asset] -= assetAmountBeforeFee;
```

```
        } else {
@>              totalAssetDepositsTracked[asset] = 0;
        }
    }
```

Recommendations

If the `totalAssetDepositsTracked` is not enough to pay the withdrawal assets, we should revert.

Low findings

[L-01] Inconsistent timestamps cause incorrect staleness checks and info

The `KHYPEPriceOracle` contract has an architectural inconsistency where dynamic price fetching from `StakingAccountant` provides fresh data but does not update the `lastUpdateTime` timestamp, causing staleness validation to use outdated timestamps and potentially reject fresh prices.

The root issue is in `getAssetPrice()`, which is a view function that fetches fresh data but cannot update state:

```
function getAssetPrice(address asset) external view override returns (uint256) {
    if (useManualPrice) {
        return manualPrice; // Uses correct timestamp from manual updates
    }

    // Fetches FRESH data from StakingAccountant but cannot update lastUpdateTime
    try stakingAccountant.kHYPEToHYPE(1e18) returns (uint256 hypeAmount) {
        uint8 kHYPEDecimals = IERC20Metadata(KHYPE_TOKEN).decimals();
        return UtilLib.scaleToEighteenDecimals(hypeAmount, kHYPEDecimals);
    }
}
```

However, staleness validation functions rely on the stale timestamp:

```
function getAssetPriceWithStalenessCheck(address asset) external view returns (uint256 price) {
    price = this.getAssetPrice(asset); // Gets fresh price from StakingAccountant

    if (maxStaleness > 0) {
        uint256 age = block.timestamp - lastUpdateTime; // Uses outdated timestamp!
        if (age > maxStaleness) {
            revert PriceTooStale(lastUpdateTime, maxStaleness); // May reject fresh data
        }
    }
}
```

This creates several problems:

(1) Fresh prices from `StakingAccountant` may be incorrectly rejected as "stale". For example, if the oracle was initialized 2 hours ago but `getAssetPrice()` fetches current data, the staleness check calculates age as 2 hours instead of recognizing the data is fresh.

(2) External consumers calling `getLastUpdateTime()` receive timestamps that don't reflect actual data freshness in dynamic mode.

The `lastUpdateTime` is only updated in non-view functions like `updateManualPrice()`, `refreshPrice()`, and `updateLastUpdateTime()`, but not during normal dynamic price fetching, creating a fundamental mismatch between data freshness and timestamp tracking.

Recommendation: Document that staleness checks only work reliably after manual `refreshPrice()` calls in dynamic pricing mode, or redesign the staleness logic to handle the architectural constraint where view functions cannot update timestamps during dynamic price fetching.

[L-02] Multiple rounding down issue for total value

In `_getTotalValueInProtocol` function, total value in protocol is calculated by looping between assets. It scales up to 18 decimals and then it multiplies by the price and removes 18 decimals from it. However following calculation method causes multiple times rounding down.

`totalValueInProtocol += totalAssetTVLScaled * assetPrice / 1e18;`.

Instead, consider removing 1e18 from here and dividing it by 1e18 at the end of loop.

[L-03] Incorrect yield tracking in case of amount > tracked allocation

In `deallocateFromStrategy` function, yield amount is tracked as `withdrawn - amount`, however, it's incorrect because tracked strategy allocation is the actual amount forwarded to the strategy, and if the amount variable is bigger than this tracking number, it will cause incorrect tracking for yield.

Consider calculating yield as `withdrawn - reductionAmount`.

[L-04] Incorrect gap slot amount due to incorrect variable counting

In many contracts, contracts layouts are trying to reach 50 variable slot by adding gap slots, however, variable countings are incorrect for some contracts. ELytraConfig contract leaves a 43-slot gap at the end of contract by assuming it has 7 variables, but it's incorrect, it has 9 variables and it should have 41 gap. ElytraDepositPoolV1 has 10 variables, but it leaves a 39-slots gap. ElytraUnstakingVaultV1 contract has 7 variables, but it leaves a 44-slots gap.

[L-05] Asymmetric yield treatment for queued withdrawals

Queued withdrawals experience **asymmetric yield treatment** compared to instant withdrawals, violating the unified pool's fairness principle. The protocol locks exchange rates at **request time** for queued users but applies **current rates** for instant withdrawers, causing queued users to forfeit yield earned during unstaking periods while receiving protection from losses.

If you look ath the **code** getAssetAmountToReceive is using the current rate in instant withdrawls, but in queued it is using the fixed rate when requesting withdrawals.

```
// INSTANT: Dynamic rate at execution
uint256 assetAmount = getAssetAmountToReceive(asset, elyAssetAmount); // ✅ Current rate

// QUEUED: Fixed rate at request
assetAmount: assetAmount, // ❌ LOCKED at request time, never updated
```

It cause issues during **yield period** -> Instant users get 110 HYPE, queued users get 100 HYPE (miss 10% yield).

And in **loss period** -> Instant users get 90 HYPE, queued users get 100 HYPE (protected from 10% loss).

This asymmetry creates **behavioral distortions** where users avoid queued withdrawals during expected yield periods, stressing protocol liquidity and contradicting liquid staking's continuous yield promise.

Recommendation

Calculate assetAmount at completion time instead of request time to ensure queued users receive current exchange rates and don't forfeit yield earned during unstaking periods.

[L-06] Inconsistent withdrawal control in `DepositPool` vs `UnstakingVault`

The `completeWithdrawal()` function in `ElytraUnstakingVaultV1` does not respect the global withdrawal controls that exist in `ElytraDepositPoolV1`. When admins disable withdrawals during emergencies, users can still complete delayed withdrawals through the unstaking vault.

Inconsistency:

```
// DepositPool - Respects withdrawal controls
function withdrawHYPE(...) external whenWithdrawalsEnabled { } ✅
function withdrawAsset(...) external whenWithdrawalsEnabled { } ✅

// UnstakingVault - Ignores withdrawal controls
function completeWithdrawal(...) external nonReentrant { } ❌
```

Problem -> Admin expects setting `withdrawalsEnabled = false` to stop ALL withdrawals, but delayed withdrawal completions continue working.

Impact -> During security incidents or oracle failures, admin cannot fully control fund outflows, undermining emergency response procedures.

Recommendation

Add withdrawal control check to completeWithdrawal function, or maybe just sync it with requestWithdrawal, which uses `whenNotPaused`.

```
function completeWithdrawal(uint256 requestId)
    external
    nonReentrant
    whenWithdrawalsEnabled  // ✅ Add unified control
{
    // existing logic...
}
```

[L-07] `getElyAssetAmountToMint()` uses outdated `elyAsset` prices, causing issues

The public view function `getElyAssetAmountToMint()` use inconsistent staleness validation, causing external integrations to receive stale price estimates that don't match actual execution results.

Inconsistent staleness validation:

```
function getAssetAmountToReceive(address asset, uint256 elyAssetAmount) public view {
    // M-01 Fix: Use staleness-checked prices for secure calculation (both in 18 decimals)
    uint256 assetPrice = oracle.getAssetPriceWithStalenessCheck(asset);  ✅ WITH staleness
    uint256 elyAssetPrice = oracle.elyAssetPrice();                      ❌ WITHOUT staleness

    return (elyAssetAmount * elyAssetPrice) / assetPrice;
}
```

in the **Actual execution it uses fresh prices** by calling updateElyAssetPrice.

```
function withdrawAsset(...) external {
    // M-02 Fix: Update elyAsset price before withdrawal
    IElytraOracle(oracleAddr).updateElyAssetPrice();  ✅ Forces fresh price

    uint256 assetAmountBeforeFee = getAssetAmountToReceive(asset, elyAssetAmount);
}
```

External protocols using stale view function estimates for slippage calculations will experience transaction reverts when elyAsset price updates between estimate and execution, causing failed integrations.

Recommendation

Use consistent staleness validation for getAssetAmountToReceive.

```
function getAssetAmountToReceive(address asset, uint256 elyAssetAmount) public view {
    uint256 assetPrice = oracle.getAssetPriceWithStalenessCheck(asset);
    uint256 elyAssetPrice = oracle.getElyAssetPriceWithStalenessCheck(); // ✅ Fixed

    return (elyAssetAmount * elyAssetPrice) / assetPrice;
}
```

This ensures external consumers get the same price validation as internal execution flows, eliminating estimate-to-execution discrepancies.

[L-08] No cap on protocol fee setting creates DoS risk

The `setProtocolFeeBps()` function in `ElytraConfigV1` lacks validation to cap the protocol fee at reasonable levels, creating inconsistency with other protocol functions and potential DoS risk.

If you check the below 2 contracts - **ElytraOracleV1**: Has 10% (1000 BPS) cap with proper validation. - **ElytraConfigV1**: No cap, allows unlimited fee values.

```
// ElytraOracleV1.sol - PROTECTED
function setProtocolFee(uint256 _feeInBPS) external override onlyElytraAdmin {
    uint256 MAX_FEE_BPS = 1000; // Max 10% fee ✅
    if (_feeInBPS > MAX_FEE_BPS) {
        revert ProtocolFeeExceedsMaximum(_feeInBPS, MAX_FEE_BPS);
    }
}

// ElytraConfigV1.sol - VULNERABLE
function setProtocolFeeBps(uint256 _protocolFeeInBPS) external onlyRole(DEFAULT_ADMIN_ROLE) {
    protocolFeeInBPS = _protocolFeeInBPS; // ❌ No validation
}
```

Even thought admin might be trusted, but setting fees above 10,000 BPS (100%) would cause arithmetic underflows in fee calculations, making all deposits/withdrawals revert and effectively DoSing the protocol until corrected.

Recommendation

Add the same fee cap validation used in `ElytraOracleV1`.

```
function setProtocolFeeBps(uint256 _protocolFeeInBPS) external onlyRole(DEFAULT_ADMIN_ROLE) {
    uint256 MAX_FEE_BPS = 1000; // Max 10% fee
    if (_protocolFeeInBPS > MAX_FEE_BPS) {
        revert ProtocolFeeExceedsMaximum(_protocolFeeInBPS, MAX_FEE_BPS);
    }
    protocolFeeInBPS = _protocolFeeInBPS;
    emit ProtocolFeeUpdated(_protocolFeeInBPS);
}
```

[L-09] Missing storage gaps in upgradeable oracle contracts

The `KHYPEPriceOracle` and `WHYPEPriceOracle` contracts are missing storage gaps despite being upgradeable (UUPS pattern). Storage gaps reserve storage slots to allow adding new state variables in future upgrades without shifting existing storage layout, preventing data corruption.

During future upgrades, adding new state variables could overwrite existing storage slots, corrupting contract state and potentially breaking oracle functionality.

Recommendation

Add storage gaps to both oracle contracts:

```
/*//////////////////////////////////////////////////////////////
                            STORAGE GAP
//////////////////////////////////////////////////////////////*/

/// @notice Storage gap to allow for future upgrades
uint256[47] private __gap;
```

This ensures the team can add new state variables in future upgrades without compromising storage compatibility.

[L-10] Inconsistency between the comment and implementation

In ElytraDepositPool, we have one variable `MIN_DUST_AMOUNT`. According to the comment, this is one amount limitation for any operation.

However, based on the current implementation, this MIN_DUST_AMOUNT will only work for the deposit function. If users want to withdraw assets, we don't have any limitations here.

```
    uint256 public constant MIN_DUST_AMOUNT = 1000; // Minimum 1000 wei for any operation
```

Recommendation: Make the comment and implementation consistent.

[L-11] Static allocation vs dynamic strategy causes DoS in loss reporting

The `reportStrategyLoss()` function validates slashing amounts against statically tracked allocations (`assetStrategyAllocations`) rather than real-time strategy balances, causing complete denial of service when strategies earn rewards before slashing events occur. This validation logic prevents legitimate slashing losses from being reported, leading to permanent accounting corruption and potential protocol insolvency.

The core issue stems from Elytra's dual tracking system: TVL calculations correctly use real-time strategy balances (fixing H-04 from the previous audit), but slashing validation still relies on stale static allocation tracking. When strategies generate yield, their real balances grow beyond the originally tracked allocation amounts, making any significant slashing event impossible to report.

```
function reportStrategyLoss(address asset, address strategy, uint256 amount) external
onlyElytraAdmin {
    // Validates against stale static tracking
    if (assetStrategyAllocations[asset][strategy] < amount) revert
LossExceedsStrategyAllocation();

    // Updates only static tracking, creating further discrepancies
    assetsAllocatedToStrategies[asset] -= amount;
    assetStrategyAllocations[asset][strategy] -= amount;
}
```

This creates a fundamental mismatch where slashing calculations are performed against real-time balances (as they should be), but validation uses outdated static allocations. The protocol includes a manual `syncStrategyAllocations()` function to reconcile discrepancies, but this creates a fragile two-step process that fails under emergency conditions.

Vulnerable Scenario: The following steps demonstrate how normal protocol operations lead to a complete slashing reporting breakdown:

1. **Initial Allocation**: Protocol allocates 100 ETH to Strategy A, tracked as `assetStrategyAllocations[WETH][StrategyA] = 100 ETH`.
2. **Strategy Growth**: Over 6 months, Strategy A earns 25% yield through normal restaking rewards, reaching 125 ETH real balance while tracked allocation remains 100 ETH.
3. **Slashing Event**: Validator experiences 20% slashing (realistic scenario), resulting in 20 ETH loss that needs to be reported.
4. **DoS Trigger**: Admin attempts `reportStrategyLoss(WETH, StrategyA, 20)` but transaction reverts because `20 ETH > 0 ETH` (20 ETH loss amount exceeds the 0 ETH growth margin between 125 ETH real balance and 100 ETH tracked allocation).
5. **Protocol Breakdown**: Slashing cannot be reported, internal accounting remains incorrect, users can withdraw at inflated rates based on pre-slash accounting, leading to potential insolvency.
6. **Emergency Complexity**: Under slashing event pressure, admin must remember to call `syncStrategyAllocations()` first, then `reportStrategyLoss()`, creating operational risk and human error potential.

This scenario becomes highly likely with even conservative parameters: 15% strategy growth over 3-6 months combined with 15-20% slashing events, both of which are realistic in liquid restaking environments.

Recommendations

Implement the following solution to eliminate the static validation dependency:

Modify `reportStrategyLoss()` to validate against the current strategy balance rather than stale allocations:

```
function reportStrategyLoss(address asset, address strategy, uint256 amount) external
onlyElytraAdmin {
    uint256 realTimeBalance = IElytraStrategy(strategy).getBalance(asset);
    require(amount <= realTimeBalance, "Loss exceeds current strategy balance");
    // Process loss reporting
}
```

[L-12] `kHYPE` on-chain price validation allows frontrun profit

The current kHYPE oracle directly uses on-chain exchange rate of kHYPE token as a price. However, `kHYPEToHYPE` function's reported value can be changed by Kinetiq's `ValidatorManager` due to the new validator profit/loss report.

```
        try stakingAccountant.kHYPEToHYPE(1e18) returns (uint256 hypeAmount) {
            // Validate the returned price is reasonable
            if (hypeAmount == 0) {
```

```
            return _handlePriceFetchFailure("Zero price returned from StakingAccountant");
        }

        // Query kHYPE token decimals and ensure proper scaling to 18 decimals
        uint8 kHYPEDecimals = IERC20Metadata(KHYPE_TOKEN).decimals();
        return UtilLib.scaleToEighteenDecimals(hypeAmount, kHYPEDecimals);
    } catch Error(string memory reason) {
        return _handlePriceFetchFailure(reason);
    } catch {
        return _handlePriceFetchFailure("Unknown error fetching from StakingAccountant");
    }
```

It's very easy to frontrun this report publication and deposit assets at a higher price or request withdraw at a lower price.

Recommendations

Consider checking kHYPE price from an external off-chain connected price oracle.

[L-13] Request withdrawals may revert due to max capital efficiency

This is actually not an implementation bug, but it's a serious design logic problem. In order to reach the best capital efficiency, the protocol should use a very high percentage of deposited assets in the strategies. However, if the protocol does that, most of the time `requestWithdrawal` function will revert due to missing liquidity in pool contract.

`if (!_hasSufficientLiquidity(asset, assetAmount)) revert InsufficientLiquidityForWithdrawal();`

In order to lower these reverts protocol should use a smaller percentage of deposited assets, but this time it will reduce the capital efficiency, which is not a good thing for any protocol.

Recommendations

Generally, this kind of situation is handled by request-withdraw cycles. User requests to withdraw, it won't fail, and it accounts for the requested amount. Later, operator brings that liquidity from strategies within the period, and the user can claim their balance after the waiting period.

In this way, capital efficiency is still very high because protocol takes liquidity from strategies only in withdrawal case, and operator can always know how much liquidity is needed for withdrawals instead of assuming.

[L-14] `assetsAllocatedToStrategies` will be incorrect after loss scenario

In `deallocateFromStrategy`, one of the possible scenarios is the loss scenario. If there is a loss withdrawn amount will be lower than the requested amount. However, if there is a loss, the reduction amount is not correct because it will substract `min(withdrawn, assetsAllocatedToStrategies[asset])`.

```
        if (withdrawn > amount) {
            // Yield scenario: reduce allocation by requested amount, yield is captured
separately
            reductionAmount = Math.min(amount, assetsAllocatedToStrategies[asset]);
        } else {
            // Loss or exact scenario: reduce allocation by actual withdrawn amount
            reductionAmount = Math.min(withdrawn, assetsAllocatedToStrategies[asset]);
        }

        // Update tracking - both values reduced by the same amount to maintain consistency
        assetsAllocatedToStrategies[asset] -= reductionAmount;
```

Actually, it should substract `min(amount, assetsAllocatedToStrategies[asset])` again in here because if there is a loss, it will subtract lower number and `assetsAllocatedToStrategies` will be inflated.

Recommendations

Consider removing the if check and calculate `reductionAmount` as `Math.min(amount, assetsAllocatedToStrategies[asset])` for both cases.

[L-15] User assets may be stolen

In Elytra, the total tvl assets are calculated based on the formula: `deposit track in pool + strategy balance + unstaking valut - reserve asset`.

In strategy, we have one `emergencyWithdraw` function. In an emergency, the admin can withdraw assets from the strategy. The `emergencyWithdraw` in the strategy contract is not exposed. There are two possible implementations from our test case.

If the actual `emergencyWithdraw` is similar to version 1, when we withdraw assets emergently, our strategy balance will become 0 immediately. This will cause share's price to drop.

If there is not one price limitation check, malicious users may deposit tiny assets to gain lots of shares.

```
    function emergencyWithdraw(address asset) external returns (uint256) {
        uint256 totalBalance = principalDeposits[asset] + accumulatedYield[asset];
        principalDeposits[asset] = 0;
        accumulatedYield[asset] = 0;
        IERC20(asset).transfer(msg.sender, totalBalance);
```

```
        return totalBalance;
    }
    function emergencyWithdraw(address asset) external returns (uint256 amount) {
        amount = IERC20(asset).balanceOf(address(this));
        if (amount > 0) {
            IERC20(asset).transfer(msg.sender, amount);
        }
        return amount;
    }
```

Recommendations

When we are in emergency mode, users should not be allowed to deposit into the pool.

[L-16] Users can frontrun the possible slash in strategy to avoid the loss

When users withdraw assets or request to withdraw assets, we will calculate the current total asset tvl and get the share's price.

In our strategy, we will stake assets with validators. The strategy will get some rewards and maybe some potential slash according to validators' behavior. The code below is a simulation of behavior from H_02.t.sol.

Users can monitor the related validators' performance, and withdraw or request withdraw before there is a possible slashing. Users can avoid the potential loss.

```
    function generateYield(address asset, uint256 yieldAmount) external {
        accumulatedYield[asset] += yieldAmount;
    }

    // Simulate slashing/loss
    function simulateSlashing(address asset, uint256 lossAmount) external {
        uint256 totalBalance = deposits[asset] + accumulatedYield[asset];
        if (lossAmount >= totalBalance) {
            deposits[asset] = 0;
            accumulatedYield[asset] = 0;
        } else {
            // Reduce from yield first, then from deposits
            // We will reduct the yield at first, then reduct the principle here.
            if (accumulatedYield[asset] >= lossAmount) {
                accumulatedYield[asset] -= lossAmount;
            } else {
                uint256 fromYield = accumulatedYield[asset];
                uint256 fromDeposits = lossAmount - fromYield;
                accumulatedYield[asset] = 0;
                deposits[asset] -= fromDeposits;
            }
        }
    }
```

Recommendations

When users request withdrawal, record the current share's price. When users finish withdrawal, we should check the latest share's price. If the latest share's price is less than the recorded share price, the users should incur some loss.

[L-17] Delayed slashing reporting enables timing-based loss avoidance

The Elytra protocol uses a manual slashing loss reporting mechanism through `reportStrategyLoss()` that creates a critical timing window where sophisticated users can front-run slashing reports to escape losses entirely, while uninformed users bear the full slashing impact. This leads to systematic wealth transfer from regular users to sophisticated actors and complete failure of fair loss distribution.

When a validator slashing event occurs, the strategy's real balance immediately drops, but Elytra's internal accounting (`assetsAllocatedToStrategies`) remains unchanged until an admin manually calls `reportStrategyLoss()` . During this gap, sophisticated users can monitor validator slashing events and withdraw at pre-slash rates, while late withdrawers face the full concentrated loss.

```
function reportStrategyLoss(address asset, address strategy, uint256 amount)
    external onlyElytraAdmin // Manual admin-only process
{
    // No automatic triggers - purely manual detection and reporting
    assetsAllocatedToStrategies[asset] -= amount;
    assetStrategyAllocations[asset][strategy] -= amount;
    emit StrategyLossReported(asset, strategy, amount);
}
```

The protocol lacks any protective mechanisms, such as withdrawal delays, automatic slashing detection, or fair queuing systems, that would prevent this timing advantage.

Vulnerable Scenario: The following steps demonstrate the systematic wealth extraction:

1. **Initial State**: Pool has 1000 ETH total - Alice holds 500 ETH shares (50%), Bob holds 500 ETH shares (50%).
2. **Slashing Event Occurs**: Validator gets slashed 20% (200 ETH lost), and the real strategy balance drops to 800 ETH immediately.
3. **Timing Window Opens**: Elytra's `assetsAllocatedToStrategies` still shows 1000 ETH, but `IElytraStrategy.getBalance()` returns 800 ETH.
4. **Alice Front-runs**: Sophisticated user Alice monitors validators, detects slashing, and immediately withdraws her 500 ETH shares at pre-slash rates, extracting full 500 ETH value.
5. **Admin Reports Loss**: Admin finally calls `reportStrategyLoss(asset, strategy, 200)` to update internal tracking to 800 ETH.

6. **Bob Bears Full Loss**: When Bob withdraws his 500 ETH shares, only 300 ETH remains in the pool - Bob loses 200 ETH while Alice escapes all losses.

7. **Wealth Transfer Complete**: Alice extracted 200 ETH extra value at Bob's expense through a timing advantage.

Recommendations

Implement one or more of the following mitigations to ensure fair loss distribution:

1. **Automated Slashing Detection**: Replace manual reporting with automated detection mechanisms that query strategy balances and immediately update internal accounting when discrepancies are detected.

2. **Withdrawal Delays**: Implement time-delayed withdrawals during slashing events, with a grace period that allows proper loss accounting before processing withdrawals.

3. **Fair Queuing System**: Process withdrawals in first-in-first-out order with locked-in exchange rates, preventing timing advantages.

The recommended approach is to implement automated detection (#1) combined with withdrawal delays (#2) to eliminate the timing window that enables this attack.

[L-18] Complete withdrawal fee bypass via unstaking vault

Users can completely avoid withdrawal fees by using the unstaking vault instead of direct withdrawals. The `ElytraDepositPoolV1` charges configurable withdrawal fees, but `ElytraUnstakingVaultV1` provides identical withdrawal functionality with zero fees.

Let's compare the code DepositPool and unstaking vault.

DepositPool (WITH fees):

```
function withdrawAsset(...) external {
    uint256 fee = Math.ceilDiv(assetAmountBeforeFee * withdrawalFee, BASIS_POINTS);
    assetAmount = assetAmountBeforeFee - fee; // ✅ Deducts fee
    IERC20(asset).safeTransfer(feeRecipient, fee); // ✅ Pays protocol
}
```

UnstakingVault (NO fees):

```
function requestWithdrawal(...) external {
    uint256 assetAmount = _calculateAssetAmount(asset, elyAssetAmount); // ❌ Full amount
    withdrawalRequests[requestId] = WithdrawalRequest({
        assetAmount: assetAmount, // ❌ No fee deducted
    });
}

function completeWithdrawal(...) external {
    IERC20(request.asset).safeTransfer(request.user, request.assetAmount); // ❌ Full amount
}
```

Users simply call `requestWithdrawal()` instead of `withdrawAsset()` to avoid fees entirely. If unstaking period is set to 0, this becomes an instant fee-free withdrawal.

let's take an **example** - Direct withdrawal: Get 990 USDC (after 1% fee on 1000 USDC). - Vault withdrawal: Get 1000 USDC (no fee). - **User saves 10 USDC per withdrawal**.

Impact -> All users will migrate to fee-free unstaking vault withdrawals, causing 100% loss of protocol withdrawal fee revenue.

Recommendation

Apply withdrawal fees in the unstaking vault to maintain consistency.

[L-19] Improper decimal process when we calculate the totalValueInProtocol

In Elytra, we can support multiple assets to stake, including native HYPE, LST, stable coins. In order to support multiple assets, we will use oracle adapters to calcualte the quote token price based on the base token.

According to ADD_ASSET.md, the oracle price will always return price with 18 decimal. Refer the doc and example, we will fetch the basic price, 1 UNIT(10**decimal) USDC's price in term of USD, then we will normalized to 18 decimal via the `SCALING_FACTOR`. The actual price value we return is 1e18 amount USDC's price in term of USD.

When we check kHYPE oracle's implementation, we have one similar implementation. We will fetch the basic price, 1 UNIT(10**decimal) kHYPE's price in term of HYPE, then we will normalize to 18 decimal.

The problem here is that the decimal's calculation is incorrect. In Hype/LST market, Hype and kHYPE are 18 decimals, so we cannot find issues here.

Let's consider stable coin market, take USDC(decimal 6) and hbUSDT(decimal 18) as example. 1. 1 USDC. 1 UNIT USDC equals 1e6 USDC, and we need to scale to 18 decimal, the return price is 1e18. `totalAssetTVLScaled * assetPrice = 1e18 * 1e18`. 2. 1 hbUSDT. 1 UNIT hbUSDT(1e18 decimal) equals 1e6 USDC and the scaled price is still 1e6. `totalAssetTVLScaled * assetPrice = 1e18 * 1e6`. Based on the above example, although 1 USDC and 1 hbUSDT should have the same value, our calculation will give out different value.

```
contract USDCPriceOracle is IPriceFetcher {
    uint256 private constant PRICE_DECIMALS = 18;
    uint256 private constant USDC_DECIMALS = 6;
    uint256 private constant SCALING_FACTOR = 10**(PRICE_DECIMALS - USDC_DECIMALS);

    function getAssetPrice(address asset) external view returns (uint256) {
        require(asset == USDC_ADDRESS, "Unsupported asset");

        // Get price from external source (e.g., Chainlink, HyperCore)
        uint256 rawPrice = _getExternalPrice();
```

```
        // Normalize to 18 decimals
        return rawPrice * SCALING_FACTOR;
    }
}

    function getAssetPrice(address asset) external view override returns (uint256) {
        try stakingAccountant.kHYPEToHYPE(1e18) returns (uint256 hypeAmount) {
            // Validate the returned price is reasonable
            if (hypeAmount == 0) {
                return _handlePriceFetchFailure("Zero price returned from StakingAccountant");
            }

            // Query kHYPE token decimals and ensure proper scaling to 18 decimals
            uint8 kHYPEDecimals = IERC20Metadata(KHYPE_TOKEN).decimals();
            return UtilLib.scaleToEighteenDecimals(hypeAmount, kHYPEDecimals);
        }
    }
    function _getTotalValueInProtocol() private view returns (uint256 totalValueInProtocol) {
@>          uint256 totalAssetTVLScaled = UtilLib.scaleToEighteenDecimals(totalAssetTVL,
assetDecimals);
@>          totalValueInProtocol += totalAssetTVLScaled * assetPrice / 1e18;
    }
```

Recommendations

Based on current price's design, we will return actual base token amount per 1e18 quote tokens. We should not use the scaled balance to calculate the value.