Pashov Audit Group

Elytra Security Review

Conducted by:
zark
unforgiven
JCN
btk

July 10th 2025 - July 16th 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

Critical findings ... 8

[C-01] `receiveFromDepositPool()` does not track assets transferred ... 8
[C-02] Withdrawal requests through deposit pool lost permanently ... 9
[C-03] TVL errors by including pending withdrawal assets ... 10

High findings ... 13

[H-01] TVL double-counts assets returned from strategy to vault ... 13
[H-02] `elyAsset` price calculation errors cause underflow reverts ... 13
[H-03] Scaling error in `elyAsset` price calculation leads to fee loss ... 15
[H-04] Strategy allocation tracking errors affect TVL calculations ... 15

Medium findings ... 19

[M-01] `_getAtomicPrices()` uses stale oracle prices without validation ... 19
[M-02] Deposit and withdrawal use stale `elyAsset` prices leading to extraction ... 20
[M-03] Deposit fees are not transferred to fee recipient in `ElytraDepositPoolV1` ... 21
[M-04] Unstaking period not snapshotted per request causing inconsistencies ... 21
[M-05] Incorrect deposit limit check order ... 22
[M-06] Cross contract reentrancy bug in `withdrawHYPE` ... 23

Low findings ... 25

[L-01] Deposit Pool may face inflation attack under certain conditions ... 25
[L-02] Missing slippage protection on withdrawals ... 27
[L-03] Unbounded growth of `withdrawalRequests` array in `ElytraUnstakingVaultV1` ... 28
[L-04] Fails to update `elyAsset` price before limit in `setPricePercentageLimit()` ... 28
[L-05] Timestamp update fails in `updateElyAssetPrice()` when `totalSupply` is zero ... 28
[L-06] Griefing attacks possible via balance manipulation in `getTotalAssetDeposits()` ... 29
[L-07] Changing asset strategy without zero allocation check ... 29
[L-08] Rounding down in fee computation enables users to bypass fees ... 30
[L-09] Inconsistent strategy allocation tracking in `deallocateFromStrategy()` ... 31
[L-10] `burnFrom` function incorrectly decreases unlimited allowances ... 31
[L-11] `grantRole` and `revokeRole` in `elyHYPE` lack admin privileges ... 32
[L-12] Upgradable smart contracts do not include a storage gap ... 33
[L-13] Deposit limit bypass due to not accounting for strategy allocations ... 33
[L-14] Asset disablement front-running causes TVL drop and holder losses ... 34
[L-15] Frequent `updateElyAssetPrice` calls bypass fee due to rounding ... 35
[L-16] `requestWithdrawal()` calculates amount but does not lock tokens ... 35
[L-17] `updateAssetStrategy` does not check allocation to strategy is zero ... 36
[L-18] Deficient validation in `updatePriceOracleForValidated` ... 37
[L-19] Zero `maxUnstakingPeriods` bypasses limit check in `setUnstakingPeriod` ... 38
[L-20] Post-fee `depositAmount` validation in `_beforeDeposit()` ... 38
[L-21] Native `HYPE` sent directly to deposit pool will be permanently locked ... 39
[L-22] Non-18 decimal tokens cause incorrect price calculations ... 40

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 **zark**, **unforgiven**, **JCN**, **btk** engaged to review **Elytra**. A total of **35** issues were uncovered.

Protocol Summary	
Project Name	Elytra
Protocol Type	Liquid Restaking
Timeline	July 10th 2025 - July 16th 2025

Review commit hash:

- f3fd3314e9cff7d74bc1edeb2e0a5b90762fbb4a (ElytraProtocol/contracts)

Fixes review commit hash:

- 0722d6f68c96a23d077ee278c4c5324643ad40ee (ElytraProtocol/contracts)

Scope

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

6. Findings

Findings count

Severity	Amount
Critical	3
High	4
Medium	6
Low	22
Total findings	**35**

Summary of findings

ID	Title	Severity	Status
[C-01]	`receiveFromDepositPool()` does not track assets transferred	Critical	**Resolved**
[C-02]	Withdrawal requests through deposit pool lost permanently	Critical	**Resolved**
[C-03]	TVL errors by including pending withdrawal assets	Critical	**Resolved**
[H-01]	TVL double-counts assets returned from strategy to vault	High	**Resolved**
[H-02]	`elyAsset` price calculation errors cause underflow reverts	High	**Resolved**
[H-03]	Scaling error in `elyAsset` price calculation leads to fee loss	High	**Resolved**
[H-04]	Strategy allocation tracking errors affect TVL calculations	High	**Resolved**
[M-01]	`_getAtomicPrices()` uses stale oracle prices without validation	Medium	**Resolved**
[M-02]	Deposit and withdrawal use stale `elyAsset` prices leading to extraction	Medium	**Resolved**
[M-03]	Deposit fees are not transferred to fee recipient in `ElytraDepositPoolV1`	Medium	**Resolved**

ID	Title	Severity	Status
[M-04]	Unstaking period not snapshotted per request causing inconsistencies	Medium	Resolved
[M-05]	Incorrect deposit limit check order	Medium	Resolved
[M-06]	Cross contract reentrancy bug in `withdrawHYPE`	Medium	Resolved
[L-01]	Deposit Pool may face inflation attack under certain conditions	Low	Resolved
[L-02]	Missing slippage protection on withdrawals	Low	Resolved
[L-03]	Unbounded growth of `withdrawalRequests` array in `ElytraUnstakingVaultV1`	Low	Resolved
[L-04]	Fails to update `elyAsset` price before limit in `setPricePercentageLimit()`	Low	Resolved
[L-05]	Timestamp update fails in `updateElyAssetPrice()` when `totalSupply` is zero	Low	Resolved
[L-06]	Griefing attacks possible via balance manipulation in `getTotalAssetDeposits()`	Low	Resolved
[L-07]	Changing asset strategy without zero allocation check	Low	Resolved
[L-08]	Rounding down in fee computation enables users to bypass fees	Low	Resolved
[L-09]	Inconsistent strategy allocation tracking in `deallocateFromStrategy()`	Low	Resolved
[L-10]	`burnFrom` function incorrectly decreases unlimited allowances	Low	Resolved
[L-11]	`grantRole` and `revokeRole` in `elyHYPE` lack admin privileges	Low	Resolved
[L-12]	Upgradable smart contracts do not include a storage gap	Low	Resolved
[L-13]	Deposit limit bypass due to not accounting for strategy allocations	Low	Resolved
[L-14]	Asset disablement front-running causes TVL drop and holder losses	Low	Resolved

ID	Title	Severity	Status
[L-15]	Frequent `updateElyAssetPrice` calls bypass fee due to rounding	Low	Resolved
[L-16]	`requestWithdrawal()` calculates amount but does not lock tokens	Low	Resolved
[L-17]	`updateAssetStrategy` does not check allocation to strategy is zero	Low	Resolved
[L-18]	Deficient validation in `updatePriceOracleForValidated`	Low	Resolved
[L-19]	Zero `maxUnstakingPeriods` bypasses limit check in `setUnstakingPeriod`	Low	Resolved
[L-20]	Post-fee `depositAmount` validation in `_beforeDeposit()`	Low	Resolved
[L-21]	Native `HYPE` sent directly to deposit pool will be permanently locked	Low	Resolved
[L-22]	Non-18 decimal tokens cause incorrect price calculations	Low	Resolved

Critical findings

[C-01] `receiveFromDepositPool()` does not track assets transferred

Severity

Impact: High

Likelihood: High

Description

The `receiveFromDepositPool()` function in `ElytraUnstakingVaultV1` contains a flaw where it attempts to calculate received assets by comparing the vault's balance before and after a transfer operation. However, this approach fails completely for all tokens because the balance comparison will always result in identical values, causing the function to calculate a received amount of 0.

```
function receiveFromDepositPool(address asset) external onlyDepositPool {
    // All assets are now ERC20 tokens (including WHYPE)
    uint256 balanceBefore = IERC20(asset).balanceOf(address(this));
    // Assets should be transferred before calling this function
    uint256 balanceAfter = IERC20(asset).balanceOf(address(this));
    uint256 received = balanceAfter - balanceBefore;
    claimableAssets[asset] += received;
    emit AssetsReceivedFromDepositPool(asset, received);
}
```

The issue occurs because the balance is checked in the same transaction, both values are identical. The calculation `received = balanceAfter - balanceBefore` always results in 0.

This creates a problem where:

1. Users request withdrawals through the deposit pool.
2. Assets are transferred to the unstaking vault via `transferAssetToUnstakingVault()`.
3. The vault calls `receiveFromDepositPool()` to track the received assets.
4. The function calculates 0 received assets due to the flawed balance comparison.
5. Users cannot claim their withdrawals because `claimableAssets` remains unchanged.

Recommendations

Modify the `receiveFromDepositPool()` function to accept an explicit `amount` parameter from the deposit pool, then directly add this amount to `claimableAssets`. This bypasses the unreliable balance comparison method entirely.

```
function receiveFromDepositPool(address asset, uint256 amount) external onlyDepositPool {
    claimableAssets[asset] += amount;
}
```

[C-02] Withdrawal requests through deposit pool lost permanently

Severity

Impact: High

Likelihood: High

Description

The `requestWithdrawal()` function in `ElytraDepositPoolV1` creates withdrawal requests that will be permanently lost due to incorrect user address tracking. When a user calls `requestWithdrawal()`, the deposit pool transfers the user's elyAsset tokens to itself and then calls the unstaking vault's `requestWithdrawal()` function:

```
    function requestWithdrawal(
        address asset,
        uint256 elyAssetAmount
    )
        external
        nonReentrant
        whenNotPaused
        onlySupportedAsset(asset)
        returns (uint256 requestId)
    {
        address unstakingVault =
elytraConfig.getContract(ElytraConstants.ELYTRA_UNSTAKING_VAULT);

        // Transfer elyAsset tokens to this contract for burning
        address elyAssetToken = elytraConfig.getElyAsset();
        IERC20(elyAssetToken).safeTransferFrom(msg.sender, address(this), elyAssetAmount);

        // Approve unstaking vault to burn tokens
        IERC20(elyAssetToken).approve(unstakingVault, elyAssetAmount);

        // Request withdrawal through unstaking vault
        requestId = IElytraUnstakingVault(unstakingVault).requestWithdrawal(asset,
elyAssetAmount);
    }
```

However, the unstaking vault records the `msg.sender` (which is the deposit pool) as the request owner instead of the original user.

```
        withdrawalRequests[requestId] = WithdrawalRequest({
            user: msg.sender,
            asset: asset,
            elyAssetAmount: elyAssetAmount,
            assetAmount: assetAmount,
            requestTime: block.timestamp,
            completed: false
        });
```

This creates an issue where:

1. User calls `ElytraDepositPoolV1.requestWithdrawal()` with their elyAsset tokens.
2. Deposit pool transfers tokens to itself and calls `UnstakingVault.requestWithdrawal()`.
3. Unstaking vault records the deposit pool address as the request owner.
4. User attempts to complete their withdrawal via `UnstakingVault.completeWithdrawal()`.
5. Transaction reverts because `msg.sender` (user) != `request.user` (deposit pool address).

The user's tokens are effectively burned during the request creation, but they can never claim their assets because the withdrawal request is owned by the deposit pool contract rather than the user.

Recommendations

Create a separate `requestWithdrawalForUser()` function in the unstaking vault that: - It is only callable by the deposit pool. - Takes an additional `user` parameter to specify the actual request owner. - Records the provided user address as the request owner instead of `msg.sender`.

[C-03] TVL errors by including pending withdrawal assets

Severity

Impact: High

Likelihood: High

Description

The `getTotalAssetTVL()` function incorrectly includes assets that are backing **pending withdrawal requests** in its TVL calculation. This results in an inflated `elyAsset` price, leading to exploitable pricing errors.

Root Issue

When a user initiates a withdrawal via `requestWithdrawal()`, their `elyAssets` are **burned immediately**, reducing total supply. However, the corresponding assets are moved to the **unstaking vault**, and are **not removed from TVL calculations**. These assets remain counted until the user completes the withdrawal.

Since price is computed as:

```
price = totalTVL / totalSupply
```

This introduces a mismatch:

- Supply goes down (due to burned `elyAssets`).
- TVL remains the same (due to unadjusted asset accounting).

This causes the `elyAsset` price to **spike artificially**, allowing users to exploit the system by:

- Burning tokens.
- Inflating price.
- Withdrawing at an unfairly high rate.

Technical Details

In `ElytraDepositPoolV1`, TVL is calculated as:

```
function getTotalAssetTVL(address asset) public view returns (uint256 totalTVL) {
    uint256 poolBalance = IERC20(asset).balanceOf(address(this));
    uint256 strategyAllocated = assetsAllocatedToStrategies[asset];
    uint256 unstakingVaultBalance = _getUnstakingVaultBalance(asset);

    return poolBalance + strategyAllocated + unstakingVaultBalance;
}
```

The problem is that `unstakingVaultBalance` is retrieved via:

```
function _getUnstakingVaultBalance(address asset) internal view returns (uint256 balance) {
    address unstakingVault = elytraConfig.getContract(ElytraConstants.ELYTRA_UNSTAKING_VAULT);
    if (unstakingVault == address(0)) {
        return 0;
    }

    try IElytraUnstakingVault(unstakingVault).getClaimableAssets(asset) returns (uint256
claimableAmount) {
        return claimableAmount;
    } catch {
        return 0;
    }
}
```

This value includes assets allocated to pending withdrawal requests — even though the corresponding shares have already been burned.

Exploitation Example

Consider the following:

1. Initial state:
2. 15e18 HYPE in pool.
3. 10e18 elyHYPE total supply.

4. Price = 1.5 HYPE.
5. User A requests withdrawal of 6e18 elyHYPE, backed by 9e18 HYPE.
6. `elyAssets` are burned.
7. 9e18 HYPE moved to unstaking vault.
8. TVL remains 15e18.
9. New supply = 4e18.
10. New price = 3.75 HYPE (inflated).
11. User B updates price, then withdraws 4e18 elyHYPE and receives **15e18 HYPE**, extracting unearned value.

This behavior can be repeated by multiple users to drain excess value.

Recommendations

There are two main approaches to resolving this issue:

Option A: Adjust TVL Calculation

Track pending withdrawals explicitly and subtract their asset backing from the result of `getTotalAssetTVL()`. This ensures the price accurately reflects only the assets backing active elyAssets.

Option B: Delay Token Burn

Defer burning of `elyAssets` until the withdrawal is actually completed (i.e., when the user calls `completeWithdrawal()`), so that TVL and supply remain aligned throughout the withdrawal lifecycle.

Both approaches aim to ensure consistent accounting of assets vs. supply to prevent price manipulation.

High findings

[H-01] TVL double-counts assets returned from strategy to vault

Severity

Impact: High

Likelihood: Medium

Description

When assets are allocated to a strategy, `ElytraDepositPoolV1.assetsAllocatedToStrategies[asset]` is incremented. Later, the strategy can return funds to the unstaking vault via:

```
/// @notice Receives assets from strategy for withdrawal processing
/// @param asset Asset address
/// @param amount Amount received
function receiveFromStrategy(address asset, uint256 amount) external onlyStrategy {
    claimableAssets[asset] += amount;
    emit AssetsReceivedFromStrategy(asset, amount);
}
```

However, this would create a double-counting problem in the TVL calculations since the same tokens would end up counted twice 1. In the deposit pool's strategy allocation (`assetsAllocatedToStrategies[asset]`). 2. In the vault's claimable assets (`claimableAssets[asset]`).

Since `getTotalAssetTVL` sums both values, the returned strategy assets inflate TVL improperly.

Recommendations

Consider decreasing the `assetsAllocatedToStrategies` and the `assetStrategyAllocations` of `ElytraDepositPoolV1`, when `receiveFromStrategy` is being triggered.

[H-02] `elyAsset` price calculation errors cause underflow reverts

Severity

Impact: Medium

Likelihood: High

Description

The `updateElyAssetPrice()` function contains a mathematical error where `protocolFeeInHYPE` is incorrectly multiplied by 1e18 before being subtracted from `totalValueInProtocol`. Since both values are already in 18-decimal precision, this unnecessary multiplication creates a massive scaling mismatch that causes the subtraction to underflow.

```
uint256 newElyAssetPrice = (totalValueInProtocol - (protocolFeeInHYPE * 1e18)) * 1e18 /
elyAssetSupply;
```

The issue occurs because:

1. `protocolFeeInHYPE` is already in 18 decimals.
2. The code multiplies `protocolFeeInHYPE` by 1e18, making it e36.

Let's take a simple example:

`elyAssetSupply` - 1,000e18 `oldElyAssetPrice` - 1.00e18 `totalValueInProtocol` - 1,050e18 `protocolFeeInBPS` - 1,000 (10%)

```
tempElyAssetPrice = totalValueInProtocol / elyAssetSupply
                  = 1050e18 / 1000e18
                  = 1.05e18 // I scaled this to fix C-03

increaseInElyAssetPrice = tempElyAssetPrice - oldElyAssetPrice
                        = 1.05e18 - 1.00e18
                        = 0.05e18

rewardInHYPE = (increaseInElyAssetPrice * elyAssetSupply) / 1e18
             = (0.05e18 * 1000e18) / 1e18
             = 50e18

protocolFeeInUSD = (rewardInUSD * protocolFeeInBPS) / 10_000
                 = (50e18 * 1000) / 10_000
                 = 5e18

newElyAssetPrice = (1050e18 - 5e18 * 1e18) * 1e18 / 1000e18
                 = (1050e18 - 5e36) * 1e18 / 1000e18
                 = underflow ❌
```

Recommendations

Fix the calculation by removing the unnecessary 1e18 multiplication:

```
newElyAssetPrice = (totalValueInProtocol - protocolFeeInHYPE) * 1e18 / elyAssetSupply;
```

[H-03] Scaling error in `elyAsset` price calculation leads to fee loss

Severity

Impact: Medium

Likelihood: High

Description

The `updateElyAssetPrice()` function in `ElytraOracleV1` contains a calculation error where the elyAsset price is computed without proper decimal scaling. The formula `totalValueInProtocol / elyAssetSupply` fails to account for the 18-decimal precision expected for price calculations.

```
        {
            uint256 tempElyAssetPrice = totalValueInProtocol / elyAssetSupply;
            if (tempElyAssetPrice > oldElyAssetPrice) {
                uint256 increaseInElyAssetPrice = tempElyAssetPrice - oldElyAssetPrice;
                uint256 rewardInHYPE = (increaseInElyAssetPrice * elyAssetSupply) / 1e18;
                protocolFeeInHYPE = (rewardInHYPE * protocolFeeInBPS) / 10_000;
            }
        }
```

This creates an issue where:

1. Both `totalValueInProtocol` and `elyAssetSupply` are in 18 decimals (e.g., 2000e18 and 1000e18).
2. The division results in a price of 2 instead of 2e18.
3. This artificially deflates the elyAsset price by a factor of 1e18.
4. The protocol permanently loses value as `tempElyAssetPrice` will always be less than `oldElyAssetPrice`.

Recommendations

Fix the price calculation by adding the missing 1e18 scaling factor:

```
tempElyAssetPrice = (totalValueInProtocol * 1e18) / elyAssetSupply
```

[H-04] Strategy allocation tracking errors affect TVL calculations

Severity

Impact: Medium

Likelihood: High

Description

The Elytra protocol tracks strategy allocations using a static variable `assetsAllocatedToStrategies[asset]`, which is manually updated during asset allocation and deallocation. However, this tracking mechanism fails to account for:

1. **Real-time yield growth inside strategies.**
2. **Realized profits or losses during deallocation.**
3. **Slashing events incurred by restaking operations.**

As a result, the `getTotalAssetTVL()` function under- or overstates actual holdings in strategies, leading to:

- Mispriced `elyAsset` tokens.
- Front-running opportunities for malicious users.
- Inability to reflect slashing-related losses.
- Potential insolvency if users withdraw more than the protocol truly owns.

Mechanism Breakdown

TVL is calculated as:

```
function getTotalAssetTVL(address asset) public view returns (uint256 totalTVL) {
    uint256 poolBalance = IERC20(asset).balanceOf(address(this));
    uint256 strategyAllocated = assetsAllocatedToStrategies[asset];
    uint256 unstakingVaultBalance = _getUnstakingVaultBalance(asset);

    return poolBalance + strategyAllocated + unstakingVaultBalance;
}
```

The issue lies in `strategyAllocated`, which is derived from:

```
// Called during allocation
assetsAllocatedToStrategies[asset] += amount;

// Called during deallocation
uint256 withdrawn = IElytraStrategy(strategy).withdraw(asset, amount);
if (withdrawn <= assetsAllocatedToStrategies[asset]) {
    assetsAllocatedToStrategies[asset] -= withdrawn;
} else {
    assetsAllocatedToStrategies[asset] = 0;
}
```

This creates three critical problems:

Yield is Ignored During Allocation Lifecycle

When a strategy generates yield, the actual balance increases (e.g., 10 → 12 tokens), but `assetsAllocatedToStrategies` remains fixed at the original 10. This underreports TVL and keeps the `elyAsset` price **deflated.**

Attack Scenario:

- User observes that strategy yield has accrued unaccounted value.
- They deposit elyAssets at a low price.
- Admin deallocates funds (suddenly realizing yield), causing TVL and price to spike.
- Attacker immediately withdraws and captures profit that wasn't rightfully theirs.

Incorrect Profit/Loss Handling on Deallocation

The `deallocateFromStrategy()` function assumes that the amount withdrawn is equivalent to the portion being deallocated. However, if the strategy has changed in value:

- **If there was a profit:** remaining allocation is overstated.
- **If there was a loss:** allocation is understated or even **permanently inflated**, since the shortfall cannot be recorded.

This causes the TVL to drift from actual holdings over time.

Example:

- Allocate 100 USDC to a strategy → TVL tracks 100
- Strategy grows to 120 USDC
- Withdraw 60 → Allocation becomes 40, but 60 remains in strategy → TVL = 40 + 60 unstated = 100 (incorrect; should be 60)

Slashing Losses Are Not Reflected

In a restaking scenario (e.g. EigenLayer), funds allocated to validators can be slashed due to misbehavior. However, the current model has no mechanism to **reduce** `assetsAllocatedToStrategies` without actually withdrawing assets — which may no longer exist post-slash.

Thus, even if a validator loses funds, the protocol keeps reporting a higher TVL and inflated `elyAsset` price.

Recommendations

Several complementary improvements are recommended:

Option A: Real-Time Balance Checks

Replace or supplement `assetsAllocatedToStrategies` with a **dynamic on-chain view**, such as:

```
IElytraStrategy(strategy).balanceOf(asset)
```

This ensures that `getTotalAssetTVL()` reflects the actual value of assets held, including yield and losses.

Option B: Accurate Profit/Loss Accounting on Deallocation

Track the originally allocated strategy shares or value, and compare it against the actual withdrawal result to update allocations correctly.

You could implement:

- `strategyReportedValue()` function for yield-aware accounting.
- `syncStrategyAllocations()` to periodically reconcile the difference.

Option C: Manual Slashing Adjustments

Introduce a mechanism to manually **reduce** `assetsAllocatedToStrategies[asset]` in the event of slashing. For example:

```
function reportStrategyLoss(address asset, uint256 amount) external onlyAdmin {
    require(assetsAllocatedToStrategies[asset] >= amount, "Exceeds tracked allocation");
    assetsAllocatedToStrategies[asset] -= amount;
}
```

Medium findings

[M-01] `_getAtomicPrices()` uses stale oracle prices without validation

Severity

Impact: Medium

Likelihood: Medium

Description

The `_getAtomicPrices()` function in `ElytraDepositPoolV1` calls `oracle.getAssetPrice()` without checking if the price data is stale. While some oracles in the system track the last update time, the `getAssetPrice()` function doesn't validate this timestamp, allowing the protocol to use outdated prices for critical calculations.

```
    function getAssetPrice(address asset) public view onlySupportedAsset(asset) returns
(uint256) {
        address oracle = assetPriceOracle[asset];
        if (oracle == address(0)) {
            revert AssetOracleNotSupported();
        }

        // Price fetchers are responsible for returning 18-decimal normalized prices
        uint256 price = IPriceFetcher(oracle).getAssetPrice(asset);

        // Apply safety checks if configured
        if (minPrice > 0 && price < minPrice) {
            revert PriceOutOfRange(price, minPrice, maxPrice);
        }

        if (maxPrice > 0 && price > maxPrice) {
            revert PriceOutOfRange(price, minPrice, maxPrice);
        }

        return price; // Already normalized to 18 decimals by price fetcher
    }
```

The issue occurs because:

1. The `_getAtomicPrices()` function calls `oracle.getAssetPrice(asset)` directly.
2. The oracle may have stale price data that hasn't been updated recently.
3. The function doesn't check the `lastUpdateTime` or use `getAssetPriceWithStalenessCheck()`.
4. Stale prices lead to incorrect elyAsset minting calculations in `_beforeDeposit()`, which creates value extraction opportunities:

The same vulnerability affects `getAssetAmountToReceive()` and other functions that rely on `_getAtomicPrices()`.

Recommendations

Replace `oracle.getAssetPrice()` calls with `oracle.getAssetPriceWithStalenessCheck()` in any functions that use `getAssetPrice()`.

[M-02] Deposit and withdrawal use stale `elyAsset` prices leading to extraction

Severity

Impact: Medium

Likelihood: Medium

Description

The `depositAsset()` and `withdrawAsset()` functions rely on the `elyAssetPrice` from the oracle without ensuring it's up-to-date. While the `updateElyAssetPrice()` function is publicly callable, there's no mechanism to guarantee it's called before price-dependent operations, allowing users to exploit stale prices for value extraction.

The issue occurs because:

1. The `getElyAssetAmountToMint()` and `getAssetAmountToReceive()` functions call `oracle.elyAssetPrice()` directly.
2. The `updateElyAssetPrice()` function is public but not automatically triggered.
3. If market conditions change significantly, the elyAsset price may become stale.
4. Users can time their deposits/withdrawals to exploit favorable outdated prices.

The same vulnerability exists in reverse - users can withdraw more assets than they should if the elyAsset price is artificially inflated due to stale data.

Recommendations

Implement automatic price updates by calling `updateElyAssetPrice()` at the beginning of `depositAsset()` and `withdrawAsset()` functions.

[M-03] Deposit fees are not transferred to fee recipient in `ElytraDepositPoolV1`

Severity

Impact: Medium

Likelihood: Medium

Description

The `ElytraDepositPoolV1` contract has an inconsistency in fee handling between deposits and withdrawals. While withdrawal fees are correctly transferred to the `feeRecipient`, deposit fees are calculated and deducted from user deposits but never actually transferred to the fee recipient.

In the `_beforeDeposit` function:

```
// Calculate fee
uint256 fee = (depositAmount * depositFee) / BASIS_POINTS;
uint256 amountAfterFee = depositAmount - fee;
```

The fee is calculated and subtracted from the deposit amount, but unlike withdrawal functions, there's no transfer of this fee to the `feeRecipient`.

Recommendations

Consider transferring the deposit fees to the `feeRecipient` in order to have consistent fee handling. This can be done by either straight up transferring the assets to the `feeRecipient`, or else it can be done through the minting the corresponding fee `elyAssets` to the `feeRecipient`.

[M-04] Unstaking period not snapshotted per request causing inconsistencies

Severity

Impact: Medium

Likelihood: Medium

Description

The vault stores a single `unstakingPeriods[asset]` that applies to all withdrawal requests. If an admin updates the period after a user's request, that change retroactively affects both pending and future requests. For example, a user who requested a 10-day unstake could be forced to wait 20 days if the period is increased, or could withdraw early if it's decreased, leading to inconsistent and unfair unlock timings.

```
    function completeWithdrawal(uint256 requestId) external nonReentrant {
        WithdrawalRequest storage request = withdrawalRequests[requestId];

         // ...

        uint256 unstakingPeriod = unstakingPeriods[request.asset];
        if (block.timestamp < request.requestTime + unstakingPeriod) {
            revert UnstakingPeriodNotComplete();
        }

        // ...
    }

    /// @notice Sets the unstaking period for an asset
    /// @param asset Asset address
    /// @param period Unstaking period in seconds (0 = instant withdrawal)
    function setUnstakingPeriod(address asset, uint256 period) external onlyElytraAdmin {
        uint256 maxPeriod = maxUnstakingPeriods[asset];
        if (maxPeriod > 0) {
            require(period <= maxPeriod, "Period exceeds maximum");
        }
        unstakingPeriods[asset] = period;
        emit UnstakingPeriodUpdated(asset, period);
    }
```

Recommendations

On each `requestWithdrawal`, capture the current unstaking period into the request struct (e.g. `request.unstakingPeriod = unstakingPeriods[asset];`) and compare `block.timestamp` against `request.requestTime + request.unstakingPeriod`.

[M-05] Incorrect deposit limit check order

Severity

Impact: Medium

Likelihood: Medium

Description

The `depositHYPE()` function wraps native `HYPE` into `WHYPE` and store it in the pool before checking if the deposit amount exceeds the current deposit limit. This means the subsequent check:

```
// Check deposit limit
if (_checkIfDepositAmountExceedsCurrentLimit(asset, depositAmount)) {
    revert MaximumDepositLimitReached();
}
```

Operates on a `balanceOf()` that already includes the newly deposited tokens, so as a result, the limit check is ineffective. In contrast, `depositAsset()` correctly performs the limit check before the transfer. This discrepancy will lead to users' deposits being reverted while they should pass.

To better understand the issue, consider this example: - Deposit pool cap: 100 WHYPE. - Current balance: 80 WHYPE. - A user deposits 15 WHYPE, which should pass, since the real limit is 100 - 80 = 20. - However, the check incorrectly does 100 - 95 = 5 and assumes the limit is only 5 WHYPE, because it has already included the 15 WHYPE "to-be-deposited" in the balance.

Recommendations

Reorder the logic in `depositHYPE()` to ensure `_beforeDeposit()` is called before wrapping the native `HYPE`, just like in `depositAsset()`. This keeps the deposit limit logic consistent and correct.

[M-06] Cross contract reentrancy bug in `withdrawHYPE`

Severity

Impact: Medium

Likelihood: Medium

Description

Users can withdraw native HYPE from the `ElytraDepositPoolV1` by calling `withdrawHYPE`, as shown here:

```
    function withdrawHYPE(
        uint256 elyAssetAmount,
        address receiver
    )
        // ...
    {
        // ...
        // Burn elyAsset tokens
        address elyAssetToken = elytraConfig.getElyAsset();
        IElyERC20(elyAssetToken).burnFrom(msg.sender, elyAssetAmount);

        // Unwrap WHYPE to native HYPE and transfer to receiver
        IWHYPE(WHYPE_TOKEN).withdraw(whypeAmount);
        (bool success,) = payable(receiver).call{ value: hypeAmount }("");
        require(success, "HYPE transfer failed");

        // Transfer fee to recipient if applicable (in WHYPE)
        if (fee > 0) {
            IERC20(WHYPE_TOKEN).safeTransfer(feeRecipient, fee);
        }

        emit AssetWithdrawal(msg.sender, WHYPE_TOKEN, elyAssetAmount, hypeAmount);
    }
```

However, because the native HYPE amount is transferred to the user **before** the fee is sent to the `feeRecipient`, a malicious user can manipulate the transaction flow and reenter the Ely system through the oracle, calling `ElytraOracleV1::updateElyAssetPrice()`. Due to incorrect TVL accounting (since elyAssets have been burned, but not all the HYPE has been transferred out), the fee amount that hasn't yet been sent out will be considered a reward. As a result, new elyAssets could be minted as protocol fees, and the updated price would decrease instead of remaining the same for this withdrawal.

For example, if a user burns 1 elyHYPE when the total supply is 10e18 and TVL is 13e18 (price = 1.3, withdrawal fee = 10%), the system will temporarily see 9e18 supply and ~11.83e18 TVL. This causes the oracle to update the price to ~1.314. The 0.13e18 fee is interpreted as a reward, and protocol fees are minted. But once the fee is finally transferred, TVL drops, and the price incorrectly drops to ~1.29 since we have minted excess `elyHYPE` for the treasury because we thought rewards were accrued. The correct price should have remained 1.3. Thus, an incorrect amount of elyAssets is minted as protocol fees, and the price becomes inconsistent.

Recommendations

It is recommended to follow the CEI pattern and also, transfer the native HYPE amounts to the user **after** the fee transfer to the `feeRecipient`.

Low findings

[L-01] Deposit Pool may face inflation attack under certain conditions

This report outlines an edge case with a considerably **low likelihood** but **potentially high impact** if exploited.

Note: *An attacker is able to drastically inflate the* `elyAssetPrice` *as the first depositor. This will block share price updates when a price deviation limit is set. However, it's important to note that, due to share price updates being decoupled from user actions, this issue will mitigate itself after an additional deposit is performed. When this occurs, the total supply of* `elyAsset` *will return to a normal amount and allow the share price to be updated again.*

The rest of this report will consider an edge case in which the `pricePercentageLimit` is set to `0`:

An attacker is able to inflate the share price by simply depositing the minimum amount required and withdrawing all but 1 wei of their shares. At this point, the total supply of `elyAsset` will be `1` and the total assets in the pool will be equal to the deposit fees taken from the attacker's initial deposit. Note that if no deposit fees are configured, the attacker can simply donate assets directly to increase `totalAssets`.

When the `ElytraOracleV1::updateElyAssetPrice` function is called, the `newElyAssetPrice` will be calculated as `totalValueInProtocol * 1e18 / elyAssetSupply`:

```
        uint256 newElyAssetPrice = (totalValueInProtocol - (protocolFeeInHYPE * 1e18)) * 1e18 /
elyAssetSupply;
```

Since `totalValueInProtocol` is denominated in 18 decimals and `elyAssetSupply == 1`, this results in the new share price being inflated by a factor of `1e18`.

At this stage the attacker would need to ensure that the depositor will receive `0` shares from their deposit by inflating the share price, such that it is greater than the `deposit * 1e18`:

```
        elyAssetToMint = (amountAfterFee * assetPrice) / elyAssetPrice;
```

It is important to note that users have the ability to specify a minimum share amount expected during deposits. If a non-zero value is supplied, then the deposit tx will revert on these lines:

```
        if (elyAssetToMint < minElyAssetExpected) {
            revert MinimumAmountToReceiveNotMet();
        }
```

However, when a minimum share amount is not specified, the user will mint `0` shares, which will give the attacker's `1 wei` of shares claim to all of the assets in the deposit pool.

Place the test below inside of the `ElytraDepositPoolV1.t.sol` directory and run with `forge test --mt test_inflation_attack`:

```
    function test_inflation_attack() public {
        // attacker deposits minimum
        address attacker = address(0xbad);
        uint256 minDeposit = elytraDepositPool.minAmountToDeposit();
        uint256 userDeposit = 10 ether; // used for later attack

        deal(WHYPE_TOKEN, attacker, minDeposit + userDeposit);

        vm.startPrank(attacker);
        IERC20(WHYPE_TOKEN).approve(address(elytraDepositPool), minDeposit);
        elytraDepositPool.depositAsset(WHYPE_TOKEN, minDeposit, 0, "test");
        vm.stopPrank();

        uint256 attackerShareBalance = elyAsset.balanceOf(attacker);

        // attacker immediately withdraws all but 1 wei of shares
        vm.prank(attacker);
        elytraDepositPool.withdrawAsset(WHYPE_TOKEN, attackerShareBalance - 1, attacker);

        assertEq(elyAsset.balanceOf(attacker), 1);
        assertEq(elyAsset.totalSupply(), 1);
        assertEq(elytraOracle.elyAssetPrice(), 1 ether);

        // share price can not be updated due to large change in price
        vm.expectRevert(abi.encodeWithSignature("ElyAssetPriceExceedsLimit()"));
        elytraOracle.updateElyAssetPrice();

        // consider case where admin allows large change in price
        vm.prank(admin);
        elytraOracle.setPricePercentageLimit(0);

        // share price update now results in extremely inflated share price
        elytraOracle.updateElyAssetPrice();
        assertGt(elytraOracle.elyAssetPrice(), 1_000_000 ether);

        // attacker donates assets directly to pool
        vm.prank(attacker);
        IERC20(WHYPE_TOKEN).transfer(address(elytraDepositPool), userDeposit);

        // share price updated
        elytraOracle.updateElyAssetPrice();

        // case 1: user deposits with min share amount specified, tx reverts
        vm.deal(user, userDeposit);

        vm.prank(user);
        vm.expectRevert(abi.encodeWithSignature("MinimumAmountToReceiveNotMet()"));
        elytraDepositPool.depositHYPE{ value: userDeposit }(1, "test"); // expects at least 1
wei of shares to be minted

        // case 2: user deposits with no share amount specified, receives 0 shares back
        vm.prank(user);
        elytraDepositPool.depositHYPE{ value: userDeposit }(0, "test");

        assertEq(IERC20(elyAsset).balanceOf(user), 0);

        // share price updated
        elytraOracle.updateElyAssetPrice();

        // attacker redeems their 1 share for all assets in pool
        vm.prank(attacker);
        elytraDepositPool.withdrawAsset(WHYPE_TOKEN, 1, attacker);

        assertEq(IERC20(WHYPE_TOKEN).balanceOf(address(elytraDepositPool)), 0);
        assertEq(elyAsset.totalSupply(), 0);
        assertGt(IERC20(WHYPE_TOKEN).balanceOf(attacker), minDeposit + userDeposit);
    }
```

Recommendations As previously mentioned, this inflated share price will mitigate itself after a subsequent deposit is performed. However, since the current implementation of the protocol technically allows for this attack to manifest as an edge case, a possible mitigation for this issue would be to mint "dead shares" during initialization.

[L-02] Missing slippage protection on withdrawals

The deposit functions in the deposit pool contract take in a `minElyAssetExpected` value that acts as slippage protection. However, the withdrawal functions do not include such protection:

```
    function withdrawAsset(
        address asset,
        uint256 elyAssetAmount,
        address receiver
    )
...

    function withdrawHYPE(
        uint256 elyAssetAmount,
        address receiver
    )
```

```
    function requestWithdrawal(
        address asset,
        uint256 elyAssetAmount
    )
```

As a result, users can receive unfavorable rates for their withdrawals due to sudden price movements and exchange rate updates.

I recommend adding a `minUnderlyingAssetExpected` parameter to the withdraw functions and validating that the asset amount transferred to the user, or pending withdrawal, is `>= minElyAssetExpected`.

[L-03] Unbounded growth of `withdrawalRequests` array in `ElytraUnstakingVaultV1`

The `withdrawalRequests` array in the `ElytraUnstakingVaultV1` contract is never cleared or reduced in size after withdrawals are processed. As a result, the array can grow indefinitely over time. This unbounded growth can lead to issues, such as increased gas costs for functions that iterate over the array (e.g., `getPendingWithdrawals()`), and may eventually cause out-of-gas errors or failed calls, especially for off-chain integrations or frontends that rely on retrieving pending withdrawals.

Recommendation

Delete fulfilled requests from the `withdrawalRequests` array.

[L-04] Fails to update `elyAsset` price before limit in `setPricePercentageLimit()`

The `setPricePercentageLimit()` function in `ElytraOracleV1` modifies the `pricePercentageLimit` without first calling `updateElyAssetPrice()` to ensure the current price is up-to-date.

```
    function setPricePercentageLimit(uint256 _limit) external override onlyElytraAdmin {
        pricePercentageLimit = _limit;
        emit PricePercentageLimitUpdated(_limit);
    }

    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);
        }
        protocolFeeInBPS = _feeInBPS;
        emit ProtocolFeeUpdated(_feeInBPS);
    }
```

Recommendations Call `updateElyAssetPrice()` at the beginning of both `setPricePercentageLimit()` and `setProtocolFee()` functions to ensure the current price is fresh before applying configuration changes.

[L-05] Timestamp update fails in `updateElyAssetPrice()` when `totalSupply` is zero

The `updateElyAssetPrice()` function in `ElytraOracleV1` contains a design where it sets the `elyAssetPrice` to `1e18` when the total supply becomes zero, but fails to update the `lastPriceUpdateTimestamp`. This creates an inconsistency where the price is modified without updating the timestamp, causing external integrations that rely on `getElyAssetPriceWithStalenessCheck()` to incorrectly revert.

```
        if (elyAssetSupply == 0) {
            elyAssetPrice = 1 ether;
            emit ElyAssetPriceUpdated(elyAssetPrice);
            return;
        }
```

The issue occurs because:

1. When `elyAssetSupply` becomes zero, the function sets `elyAssetPrice = 1e18`.
2. However, it doesn't update `lastPriceUpdateTimestamp` to the current block timestamp.
3. External systems calling `getElyAssetPriceWithStalenessCheck()` will see an outdated timestamp and will incorrectly determine the price is stale and revert transactions.

Recommendations Update the `lastPriceUpdateTimestamp` whenever the `elyAssetPrice` is modified, including when it's set to `1e18` due to zero total supply:

```
elyAssetPrice = 1e18;
lastPriceUpdateTimestamp = block.timestamp;
```

[L-06] Griefing attacks possible via balance manipulation in `getTotalAssetDeposits()`

The `getTotalAssetDeposits()` function calculates the total asset deposits by reading the current balance of the contract using `IERC20(asset).balanceOf(address(this))`. This creates a vulnerability where malicious users can grief legitimate depositors by direct token transfers.

```
    function getTotalAssetDeposits(address asset) public view override returns (uint256
totalAssetDeposit) {
        // All assets are now ERC20 tokens (including WHYPE)
        return IERC20(asset).balanceOf(address(this));
    }
```

For example: - Current total deposits: 900 tokens. - Deposit limit: 1000 tokens. - Alice wants to deposit 100 tokens (which should be allowed). - Bob transfers 1 wei to the contract. - Alice's 100 token deposit will be rejected.

Recommendations Store the total asset deposits in a separate storage variable.

[L-07] Changing asset strategy without zero allocation check

The `updateAssetStrategy` function allows updating an asset’s strategy without verifying that the current allocation to the existing strategy is zero. This can lead to inconsistencies in asset allocation tracking and potential loss or mismanagement of funds.

It is recommended to add a check ensuring the asset’s allocation to the current strategy is zero before permitting a strategy update. Alternatively, require the deallocation of assets before changing the strategy to maintain consistency and prevent errors.

```
function updateAssetStrategy(
        address asset,
        address strategy
    )
        external
        onlyRole(DEFAULT_ADMIN_ROLE)
        onlySupportedAsset(asset)
    {
        UtilLib.checkNonZeroAddress(strategy);
        if (assetStrategy[asset] == strategy) {
            revert ValueAlreadyInUse();
        }

        assetStrategy[asset] = strategy;
        emit AssetStrategyUpdate(asset, strategy);
    }
```

[L-08] Rounding down in fee computation enables users to bypass fees

In functions like `withdrawAsset()`, `withdrawHYPE()`, and `_beforeDeposit()`, the fee calculation is performed using integer division, which rounds down the result. As a result, when the calculated fee is very small, it can become zero after rounding, allowing users to avoid paying any fee at all.

```
function withdrawAsset(
        address asset,
        uint256 elyAssetAmount,
        address receiver
    )
        --Snipped--

        // Calculate withdrawal fee
        uint256 fee = (assetAmountBeforeFee * withdrawalFee) / BASIS_POINTS;
        --Snipped--
    }
```

This behavior may be exploitable by attackers who repeatedly perform small transactions to bypass the intended protocol fees.

Recommendations

Round fee calculations up instead of down to ensure that even small operations incur a minimum fee and prevent fee avoidance. This helps protect protocol revenue and ensures fairness across all users.

[L-09] Inconsistent strategy allocation tracking in `deallocateFromStrategy()`

The `deallocateFromStrategy()` function may introduce discrepancies between `assetsAllocatedToStrategies[asset]` and the sum of all `assetStrategyAllocations[asset][strategy]`. When either of the two conditionals results in setting the value to zero (i.e., `withdrawn > tracked amount`), the values become inconsistent and desynchronized.

This discrepancy could lead to inaccurate TVL reporting or flawed internal accounting, which may affect future allocation or deallocation decisions.

```
    function deallocateFromStrategy(
        address asset,
        uint256 amount
    )
        external
        nonReentrant
        onlyElytraOperator
        onlySupportedAsset(asset)
    {
        address strategy = elytraConfig.assetStrategy(asset);
        require(strategy != address(0), "No strategy set for asset");

        // Withdraw from strategy
        uint256 withdrawn = IElytraStrategy(strategy).withdraw(asset, amount);

        // Update tracking (ensure we don't underflow)
        if (withdrawn <= assetsAllocatedToStrategies[asset]) {
            assetsAllocatedToStrategies[asset] -= withdrawn;
        } else {
@>          assetsAllocatedToStrategies[asset] = 0;
        }

        if (withdrawn <= assetStrategyAllocations[asset][strategy]) {
            assetStrategyAllocations[asset][strategy] -= withdrawn;
        } else {
@>          assetStrategyAllocations[asset][strategy] = 0;
        }

        emit AssetDeallocatedFromStrategy(asset, strategy, withdrawn);
    }
```

[L-10] `burnFrom` function incorrectly decreases unlimited allowances

The `burnFrom` function in `ElyAsset.sol` deviates from standard ERC20 behaviour by decreasing allowances even when they are set to `type(uint256).max`, which conventionally represents unlimited approval.

```
    /// @notice Burns tokens from an address if approved
    /// @param from Address to burn from
    /// @param amount Amount to burn
    function burnFrom(address from, uint256 amount) external override {
        if (msg.sender != from && !_hasBurnerRole(msg.sender)) {
            uint256 currentAllowance = allowance(from, msg.sender);
            if (currentAllowance < amount) {
                revert InsufficientAllowance(currentAllowance, amount);
            }
            _approve(from, msg.sender, currentAllowance - amount);
        }
        _burn(from, amount);
    }
```

In standard ERC20 implementations (including OpenZeppelin's), when an allowance is set to `type(uint256).max`, it represents unlimited approval and should not be decreased during transfers or burns. In order to follow the ERC20 standard correctly, consider not decreasing the allowance if it is `type(uint256).max`.

[L-11] `grantRole` and `revokeRole` in `elyHYPE` lack admin privileges

The `grantRole` and `revokeRole` functions in the `elyHYPE` contract are designed to perform role management calls to the `elytraConfig` contract, but they will always fail due to a fundamental access control issue.

```
    /// @notice Grants a role to an address
    /// @param role Role to grant
    /// @param account Address to grant role to
    function grantRole(bytes32 role, address account) public override onlyElytraAdmin {
        IAccessControl(address(elytraConfig)).grantRole(role, account);
    }

    /// @notice Revokes a role from an address
    /// @param role Role to revoke
    /// @param account Address to revoke role from
    function revokeRole(bytes32 role, address account) public override onlyElytraAdmin {
        IAccessControl(address(elytraConfig)).revokeRole(role, account);
    }
```

The issue occurs because: - The `onlyElytraAdmin` modifier verifies that the caller has admin privileges in `elytraConfig`. - The function then calls `elytraConfig.grantRole()` where `msg.sender` becomes the `elyHYPE` contract address. - The `elytraConfig.grantRole()` function (using OpenZeppelin's `AccessControl`) requires the caller to have admin privileges. - However, the `elyHYPE` contract itself was never granted admin role in `elytraConfig`.

This results in all calls to these functions reverting, making them completely non-functional code. To fix it, consider removing them or, alternatively and more dangerously, you can give admin privileges to all `elyAssets`.

[L-12] Upgradable smart contracts do not include a storage gap

Each of the contracts inherits from OpenZeppelin's `UUPSUpgradeable`, but also declares its own state variables without reserving any additional storage slots for future upgrades. Without a `uint256[50] private __gap;` at the end of your implementation contracts, adding new variables in later versions will shift the existing storage layout and can corrupt state. Take a look at this.

Consider implementing a storage gap :

```
uint256[49] private __gap;
```

[L-13] Deposit limit bypass due to not accounting for strategy allocations

The `getAssetCurrentLimit()` function in the `ElytraDepositPoolV1` contract is intended to enforce a maximum deposit limit for each asset by returning the remaining capacity for new deposits. However, the current implementation only considers the asset balance held directly by the pool (`IERC20(asset).balanceOf(address(this))`) and does not include assets that have been allocated to strategies via `assetsAllocatedToStrategies`. As a result, the protocol may allow deposits that exceed the intended limit.

For example:

1. The deposit limit for an asset is set to 1000.
2. The pool currently holds 100 of the asset, and 900 have been allocated to a strategy (tracked in `assetsAllocatedToStrategies[asset]`).
3. A user calls `getAssetCurrentLimit(asset)`, which returns 900 (1,000 - 100), ignoring the 900 already allocated to the strategy.
4. The user is able to deposit up to 900 more, resulting in a total of 1,900 assets in the system, far exceeding the intended limit.

Relevant code:

```
function getAssetCurrentLimit(address asset) public view override returns (uint256
currentLimit) {
    uint256 totalAssetDeposits = getTotalAssetDeposits(asset);
    uint256 depositLimit = elytraConfig.depositLimitByAsset(asset);

    if (totalAssetDeposits >= depositLimit) {
        return 0;
    }

    return depositLimit - totalAssetDeposits;
}
```

Recommendations

Update `getAssetCurrentLimit()` to include both the pool balance and the amount allocated to strategies (i.e., `assetsAllocatedToStrategies[asset]`) when calculating the total asset deposits.

[L-14] Asset disablement front-running causes TVL drop and holder losses

The protocol allows the admin to disable specific assets via `updateAssetSupport()` in `ElytraConfigV1` contract, which removes the asset from being considered in TVL calculations. However, this action can be front-run by an attacker who deposits the soon-to-be-disabled asset just before the admin transaction is mined.

```
    function updateAssetSupport(address asset, bool isSupported) external
onlyRole(DEFAULT_ADMIN_ROLE) {
        assets[asset].isSupported = isSupported;
        if (isSupported) {
            supportedAssets.add(asset);
        } else {
            supportedAssets.remove(asset);
        }
        emit AssetSupportUpdated(asset, isSupported);
    }
```

```
    function _getTotalValueInProtocol() private returns (uint256 totalValueInProtocol) {
        --Snipped--
@>      for (uint16 assetIdx; assetIdx < supportedAssetCount;) { //@audit: only suppoerted
assets are considered in TVL
            --Snipped--
            totalValueInProtocol += totalAssetTVL * assetPrice / 1e18;
            --Snipped--
        }
    }
```

Once the admin's action is executed, the asset is no longer counted toward TVL, but the total supply is increased, causing a drop in elyAsset price. As a result, all other users holding elyAsset tokens suffer losses due to dilution and price suppression caused by the attacker's front-running.

Recommendations

Make `updateAssetSupport()` an atomic operation that includes logic to prevent further deposits of the target asset in the same block or beforehand. Alternatively, introduce a delay or queuing.

[L-15] Frequent `updateElyAssetPrice` calls bypass fee due to rounding

The `updateElyAssetPrice()` function in the `ElytraDepositPoolV1` contract calculates the protocol fee based on the increase in asset value. This fee is first computed in HYPE, then converted into elyAsset tokens to be minted to the treasury.

```
function updateElyAssetPrice() external nonReentrant {
    --Snipped--

    // Only calculate protocol fee if newElyAssetPrice is not zero
    if (newElyAssetPrice > 0) {
        elyAssetAmountToMintAsProtocolFee = (protocolFeeInHYPE * 1e18) / newElyAssetPrice;
    }
   --Snipped--
}
```

However, due to rounding down during the conversion process, the resulting fee amount in elyAsset can be zero if the calculated HYPE fee is very small. An attacker can exploit this by calling `updateElyAssetPrice()` frequently with small increments in asset value, effectively avoiding the minting of any protocol fee.

This allows users to circumvent the intended protocol fee mechanism over time, leading to under-collection of fees and potentially undermining the revenue model of the protocol.

Recommendations

Round up the fee calculation during conversion from HYPE to elyAsset tokens to ensure the protocol always collects a minimum fee when applicable. This prevents the fee from being rounded down to zero due to small price changes.

[L-16] `requestWithdrawal()` calculates amount but does not lock tokens

In `ElytraUnstakingVaultV1`, the `requestWithdrawal()` function calculates the amount of underlying assets owed to the user based on the elyAsset's current price at the time of the request. However, it does not lock or reserve the corresponding tokens.

```
function requestWithdrawal(
    address asset,
    uint256 elyAssetAmount
)
    external
    nonReentrant
    whenNotPaused
    returns (uint256 requestId)
{
    if (elyAssetAmount == 0) revert WithdrawalNotFound();

    // Calculate asset amount using oracle
@1>     uint256 assetAmount = _calculateAssetAmount(asset, elyAssetAmount);

        // Burn elyAsset tokens immediately
        address elyAssetToken = elytraConfig.getElyAsset();
        IElyERC20(elyAssetToken).burnFrom(msg.sender, elyAssetAmount);

        // Create withdrawal request
        requestId = nextRequestId++;
        withdrawalRequests[requestId] = WithdrawalRequest({
            user: msg.sender,
            asset: asset,
@2>         elyAssetAmount: elyAssetAmount,
            assetAmount: assetAmount,
            requestTime: block.timestamp,
            completed: false
        });

        userWithdrawals[msg.sender].push(requestId);

        emit WithdrawalRequested(msg.sender, asset, elyAssetAmount, assetAmount, requestId);
    }
```

As a result, if the elyAsset's price (or TVL) declines between the request and the finalization time, there may not be enough tokens to fulfill the withdrawal. Consider the worst case scenario where a user requests to withdraw 100% of `elyAsset` tokens. Even a small decline in TVL or price during the unstaking period can cause the vault to fall short of the promised amount, making the withdrawal impossible to finalize.

This creates reliability issues and potentially traps users in a state where they cannot reclaim their funds due to price volatility or shifts in protocol balance.

Recommendations

Introduce a mechanism to lock or reserve the underlying token amount at the time of withdrawal request to guarantee availability during finalization. Alternatively, calculate the asset amounts on withdraw finalization.

[L-17] `updateAssetStrategy` does not check allocation to strategy is zero

The `updateAssetStrategy` function in `ElytraConfigV1` contract allows the protocol to assign a new strategy to an asset. However, it does not verify that the asset's allocation in the current (old) strategy is zero before making this change.

```
    function updateAssetStrategy(
        address asset,
        address strategy
    )
        external
        onlyRole(DEFAULT_ADMIN_ROLE)
        onlySupportedAsset(asset)
    {
        UtilLib.checkNonZeroAddress(strategy);
        if (assetStrategy[asset] == strategy) {
            revert ValueAlreadyInUse();
        }

        assetStrategy[asset] = strategy;
        emit AssetStrategyUpdate(asset, strategy);
    }
```

If there are still funds allocated in the old strategy at the time of the switch, those funds may become inaccessible. This is because `deallocateFromStrategy` function operates only on the **current** strategy. As a result, after the update, the admin loses the ability to deallocate or withdraw funds from the previous strategy through standard protocol mechanisms.

This can lead to funds being locked in the old strategy, causing accounting issues, capital inefficiencies, or even permanent fund loss if no emergency or manual recovery method is available.

Recommendations

Enforce a check in `updateAssetStrategy` to ensure that the current strategy has zero allocation for the asset before allowing a strategy change. Alternatively, provide a clear mechanism to migrate or recover residual funds from the old strategy to prevent lock-in and maintain operational integrity.

[L-18] Deficient validation in `updatePriceOracleForValidated`

The goal of `updatePriceOracleForValidated` in `ElytraOracleV1` is to update the oracle for a supported asset, but, also perform validations checks for it, as we can see here :

```
    /// @notice Updates the price oracle for an asset with validation
    /// @param asset Asset address
    /// @param priceOracle Oracle address
    function updatePriceOracleForValidated(
        address asset,
        address priceOracle
    )
        // ...
    {
        UtilLib.checkNonZeroAddress(priceOracle);

        // Sanity check that oracle has reasonable precision
        uint256 price = IPriceFetcher(priceOracle).getAssetPrice(asset);
        if (price > 1e19 || price < 1e17) {
            revert InvalidPriceOracle();
        }

        assetPriceOracle[asset] = priceOracle;
        emit AssetPriceOracleUpdate(asset, priceOracle);
    }
```

However, the checks are incorrect and deficient since hard-coded bounds (1e17–1e19) are used instead of the protocol's configured `minPrice` / `maxPrice`, and it does not verify the oracle's last update timestamp against `maxAssetPriceAge`. As a result, a malicious or stale price feed could be accepted, leading to mispriced deposits or withdrawals.

Recommendations

Consider validating against the contract's `minPrice` / `maxPrice` parameters instead of fixed constants, and require that the fetched price's age (via `getLastUpdateTime`) not exceed `maxAssetPriceAge` before accepting a new oracle address, since this is the goal of the `updatePriceOracleForValidated` function.

[L-19] Zero `maxUnstakingPeriods` bypasses limit check in `setUnstakingPeriod`

In `ElytraUnstakingVaultV1`, when the admin wants to enforce instant withdrawal completion, they set `unstakingPeriods[asset] = 0`:

```
    /// @notice Sets the unstaking period for an asset
    /// @param asset Asset address
    /// @param period Unstaking period in seconds (0 = instant withdrawal)
    function setUnstakingPeriod(address asset, uint256 period) external onlyElytraAdmin {
        uint256 maxPeriod = maxUnstakingPeriods[asset];
        if (maxPeriod > 0) {
            require(period <= maxPeriod, "Period exceeds maximum");
        }
        unstakingPeriods[asset] = period;
        emit UnstakingPeriodUpdated(asset, period);
    }
```

There is also logic allowing the admin to set a cap (`maxUnstakingPeriods[asset]`) on allowed ranges. However, if that cap is zero, the check is skipped entirely, so instead of restricting to instant-only, it removes all limits, permitting any period. The correct logic would be that if `maxUnstakingPeriods[asset] == 0`, only `period == 0` is allowed.

Recommendations

Consider treating 0 as a valid "instant-only" setting by requiring `period == 0` when `maxUnstakingPeriods[asset] == 0`.

[L-20] Post-fee `depositAmount` validation in `_beforeDeposit()`

In `_beforeDeposit()`, the contract validates the raw `depositAmount` against `minAmountToDeposit`, `DUST_AMOUNT`, and zero-amount checks before deducting the deposit fee. As a result, a user could pass the pre-fee threshold but, after fee deduction, end up with an effective deposit below the intended minimum or dust limits.

```
    if (depositAmount == 0) {
        revert InvalidAmountToDeposit();
    }

    // Simple minimum check - assume all amounts are in 18 decimals
    if (depositAmount < minAmountToDeposit) {
        revert InvalidAmountToDeposit();
    }

    // Prevent dust attacks
    if (depositAmount < MIN_DUST_AMOUNT) {
        revert DustAmountTooSmall();
    }
```

Recommendations

Reorder the validation steps so that the fee is subtracted first, then apply: 1. `depositAmountAfterFee > 0`. 2. `depositAmountAfterFee >= minAmountToDeposit`. 3. `depositAmountAfterFee >= MIN_DUST_AMOUNT`.

Leave the `_checkIfDepositAmountExceedsCurrentLimit` check on the pre-fee amount, since it's intended to limit gross inflows.

[L-21] Native `HYPE` sent directly to deposit pool will be permanently locked

The contract includes a `receive()` function, allowing it to accept native `HYPE`. However, there is no restriction on who can send native `HYPE` to the pool. This creates a scenario where any address can transfer native `HYPE` to the deposit pool, even outside of the intended `withdrawHYPE()` flow.

```
    /*//////////////////////////////////////////////////////////////
                            RECEIVE FUNCTIONS
    //////////////////////////////////////////////////////////////*/

    receive() external payable { }
```

Since the contract only handles native `HYPE` via wrapping/unwrapping with `IWHYPE` during deposit/withdrawal logic, any direct native `HYPE` sent to the contract via `receive()` will not be tracked, refunded, or claimable, and effectively it will lead to locking those tokens forever.

Recommendations

Restrict the `receive()` function to only accept native `HYPE` from the `WHYPE` contract. This ensures that only legitimate unwrapping operations trigger native HYPE transfers, and any accidental or malicious transfers from other addresses are reverted.

[L-22] Non-18 decimal tokens cause incorrect price calculations

The `ElytraDepositPoolV1` contract is designed to handle tokens with different decimal places but lacks any mechanism to properly scale non-18 decimal tokens. This creates an issue where tokens with fewer decimals (like USDC with 6 decimals) are treated as if they have 18 decimals, leading to massive price calculation errors.

```
        uint256 elyAssetToMint = _beforeDeposit(asset, depositAmount, minElyAssetExpected);
```

```
        // Calculate elyAsset amount (assume all 18 decimals)
        return (amount * assetPrice) / elyAssetPrice;
```

The problem occurs because:

1. The `depositAsset()` function accepts `depositAmount` in the token's native decimals.
2. The `_beforeDeposit()` function processes this amount without any decimal scaling.
3. Price calculations in `getElyAssetAmountToMint()` assume all amounts are in 18 decimals.
4. For a 6-decimal token like USDC, a deposit of 1,000,000 USDC (1M tokens) is treated as 1e12 instead of 1,000,000e18.

This creates severe value extraction scenarios:

1. User deposits 1,000,000 USDC (1M tokens with 6 decimals).
2. The system treats this as 1e12 instead of 1,000,000e18.
3. The user receives elyAsset tokens worth 1e12 times less than they should.

The same issue affects all price calculations throughout the system, including `getAssetAmountToReceive()`, `getTotalAssetTVL()`.

Recommendations

Implement decimal scaling by:

1. Adding a function to get token decimals: `IERC20(asset).decimals()`.
2. Scaling `depositAmount` to 18 decimals in `_beforeDeposit()`: `depositAmount * (10 ** (18 - tokenDecimals))`.
3. Applying the same scaling logic to all price calculation functions.