Astrolab Security Review

Pashov Audit Group

Conducted by: unforgiven, __141345__, ast3ros

January 25th 2024 - January 31st 2024

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of multiple teams of some of the best smart contract security researchers in the space. Having a combined reported security vulnerabilities count of over 1000, the group strives to create the absolute very best audit journey possible - although 100% security can never be guaranteed, we do guarantee the best efforts of our experienced researchers for your blockchain protocol. Check 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. Introduction

A time-boxed security review of the **strats** repository was done by **Pashov Audit Group**, with a focus on the security aspects of the application's smart contracts implementation.

4. About Astrolab

Astrolab DAO aims to redefine yield aggregation by providing superior yields at scale. The way to do so is through cross-chain diversification and maximized capital efficiency, leveraging top-tier liquidity aggregators on-chain and algorithmic execution.

5. Risk Classification

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

5.1. Impact

- High - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- Medium - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- Low - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

5.2. 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.

5.3. Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

6. Security Assessment Summary

review commit hash **- afe24b344f9e25104cbdd240d5d4448f01b06e07**

fixes review commit hash **- f34c311f3c5fe6adac4522a9f2a9ead75b8d639a**

Scope

The following smart contracts were in scope of the audit:

- `abstract/AsAccessControl`
- `abstract/AsRescuable`
- `abstract/StrategyV5Abstract`
- `abstract/As4626`
- `abstract/StrategyV5`
- `abstract/StrategyV5Chainlink`
- `abstract/AsManageable`
- `abstract/StrategyV5Agent`
- `abstract/StrategyV5Pyth`
- `abstract/As4626Abstract`
- `libs/AsAccounting`

7. Executive Summary

Over the course of the security review, unforgiven, __141345__, ast3ros engaged with AstrolabDAO to review Astrolab. In this period of time a total of **19** issues were uncovered.

Protocol Summary

Protocol Name	Astrolab
Repository	https://github.com/AstrolabDAO/strats
Date	January 25th 2024 - January 31st 2024
Protocol Type	Yield aggregator

Findings Count

Severity	**Amount**
Critical	2
High	7
Medium	8
Low	2
Total Findings	**19**

Summary of Findings

ID	Title	Severity	Status
[C-01]	Inflating redemption request and DoS in the vault	Critical	Resolved
[C-02]	Wrong usage of mapping target in cancelRedeemRequest	Critical	Resolved
[H-01]	Accounts not properly removed from roles upon revoking	High	Resolved
[H-02]	Flash loan wrong balance check	High	Resolved
[H-03]	Flash loan not working due to transferFrom Issue	High	Resolved
[H-04]	withdraw() worst price will distort sharePrice()	High	Resolved
[H-05]	Fee calculation mismatch in mint, deposit, redeem and withdraw	High	Resolved
[H-06]	Fee target mismatch in deposit, mint, withdraw, redeem and preview*** methods	High	Resolved
[H-07]	Accounting issues after underlying asset change	High	Resolved
[M-01]	Redeem function active when vault is paused	Medium	Resolved
[M-02]	Wrong rounding direction in previewWithdraw()	Medium	Resolved
[M-03]	Wrong rounding direction in previewMint()	Medium	Resolved
[M-04]	Wrong usage of revBp() in deposit() in fee calculation	Medium	Resolved
[M-05]	Using deprecated function in Chainlink	Medium	Resolved

[M-06]	Fee on Transfer Token Will Break accounting	Medium	Resolved
[M-07]	Using stale price in Pyth Network	Medium	Resolved
[M-08]	ERC20::approve will revert for some non-standard tokens like USDT	Medium	Resolved
[L-01]	Inconsistency in access control for setInputs	Low	Resolved
[L-02]	Storage slot collision due to adding rescueRequests in StrategyV5Agent	Low	Resolved

8. Findings

8.1. Critical Findings

[C-01] Inflating redemption request and DoS in the vault

Severity

Impact: High, because this can disrupt all primary functionalities of the vaults, leading to a denial of service

Likelihood: High, because any vault share owner can exploit this easily

Description

In `requestRedeem`, the function ensures that the shares requested for redemption are not greater than the owner's share balance. However, the operator parameter is not validated. This allows users to inflate the `req.totalRedemption` value by repeatedly submitting redemption requests with different operator addresses. The `req.totalRedemption` can be inflated to be larger than `totalAssets`.

```
function requestRedeem(
        uint256 shares,
        address operator,
        address owner
    ) public nonReentrant {
        if (owner != msg.sender || shares == 0 || balanceOf(owner) < shares)
            revert Unauthorized();
```

This inflation of `totalRedemption` subsequently leads to an increase in `pendingRedemption` during the liquidate function call by the keeper, which in turn inflates `req.totalClaimableRedemption`

```
uint256 pendingRedemption = totalPendingRedemptionRequest();

    req.totalClaimableRedemption += pendingRedemption;
```

The inflated `req.totalClaimableRedemption` disrupts the `sharePrice()` function, causing it to consistently revert. This is problematic as critical operations like deposit, mint, withdraw, and redeem rely on the `sharePrice` function.

```
function sharePrice() public view virtual returns (uint256) {
        uint256 supply = totalAccountedSupply();
        return
            supply == 0
                ? weiPerShare
                : totalAccountedAssets().mulDiv( // eg. e6
                    weiPerShare ** 2, // 1e8*2
                    supply * weiPerAsset
                ); // eg. (1e6+1e8+1e8)-(1e8+1e6)
    }
```

The sharePrice function reverts because the `totalAccountedAssets` calculation returns a negative value due to `totalAssets` being smaller than the inflated req.`totalClaimableRedemption`. This calculation error results in a DoS for the entire protocol.

```
function totalAccountedAssets() public view returns (uint256) {
        return
            totalAssets() -
            req.totalClaimableRedemption.mulDiv(
                last.sharePrice * weiPerAsset,
                weiPerShare ** 2
            ); // eg. (1e8+1e8+1e6)-(1e8+1e8) = 1e6
    }
```

Recommendations

To validate the `operator` in the `requestRedeem` function and ensure that an owner can only request redemption once, using their own shares.

[C-02] Wrong usage of mapping target in ``cancelRedeemRequest``

Severity

Impact: High, one can burn others' tokens

Likelihood: High, it can be done permissionless

Description

Users can call `cancelRedeemRequest()` can cancel their redeem requests and code would burn the excess shares from the loss incurred while not farming with the idle funds (opportunity cost). The issue is that code doesn't check that operator have allowance over owner's funds and one can call this function and burn others tokens.

also the second mistake is that code uses `req.byOperator[operator]` instead of the `req.byOperator[owner]`.

This is the POC:

1. attacker will create a redeem request when the price in the contract is 1.5 for hist ADD1 address and AMOUNT1.
2. after some time attacker that price() in the contract is 2 attacker can call `cancelRedeemRequest(owner=target_user, operator=ADD1)` with his ADD1 address.
3. now because `msg.sender == operator` code would allow the call and also there is no allowance check for owner and operator.
4. in the end because price has increased from 1.5 to 2 code would burn some of owner's shares.

```
function cancelRedeemRequest(
        address operator,
        address owner
    ) external nonReentrant {

        if (owner != msg.sender && operator != msg.sender)
            revert Unauthorized();

        Erc7540Request storage request = req.byOperator[operator];
        uint256 shares = request.shares;

        if (shares == 0) revert AmountTooLow(0);

        last.sharePrice = sharePrice();

        if (last.sharePrice > request.sharePrice) {
            // burn the excess shares from the loss incurred while not farming
            // with the idle funds (opportunity cost)
            uint256 opportunityCost = shares.mulDiv(
                last.sharePrice - request.sharePrice,
                weiPerShare
            ); // eg. 1e8+1e8-1e8 = 1e8
            _burn(owner, opportunityCost);
        }
```

There is similar bug in the `requestRedeem()` and `_withdraw()` that require attention too.

Recommendations

In `cancelRedeemRequest()` use `req.byOperator[owner]` and also make sure operator have allowance over owner tokens. In `requestRedeem()` code should save information in `req.byOperator[owner]` and also confirms that operator have allowance over owner funds. Use `req.byOperator[_owner]` in _withdraw().

8.2. High Findings

[H-01] Accounts not properly removed from roles upon revoking

Severity

Impact: High - Accounts that are removed from a role, particularly critical roles like the default admin, retain access to the role's privileges. This poses significant security risks.

Likelihood: Medium - The issue occurs consistently every time the `revokeRole` function is called.

Description

When an account is revoked from a role, `_revokeRole` function removes account from the members set.

```
function _revokeRole(bytes32 role, address account) internal virtual {
        if (hasRole(role, account)) {
            _roles[role].members.remove(account.toBytes32());
            emit RoleRevoked(role, account, msg.sender);
        }
    }
```

In the `AsSequentialSet.sol` library, roles are stored in the `Set` struct, which contains an array `data` and a mapping `index` from bytes32 to uint32.

```
struct Set {
        bytes32[] data;
        mapping(bytes32 => uint32) index;
    }
```

The `remove` function in the library is supposed to handle the removal of elements but calls `removeAt` which only removes the account from the `Set.data` array and does not reset the `Set.index`.

```
function remove(Set storage q, bytes32 o) internal {
        uint32 i = q.index[o];
        require(i > 0, "Element not found");
        removeAt(q, i - 1);
    }

    function removeAt(Set storage q, uint256 i) internal {
        require(i < q.data.length, "Index out of bounds");
        if (i < q.data.length - 1) {
            delete q.data[i];
            q.data[i] = q.data[q.data.length - 1];
        }
        q.data.pop();
    }
```

Therefore, when the hasRole function checks if a user has a role by using the `has` function.

```
function hasRole
      (bytes32 role, address account) public view virtual returns (bool) {
        return _roles[role].members.has(account.toBytes32());
    }
```

And the `has` function only checks the index of that account, and the index still exists. It means after the user is removed from the role, it still has the role.

```
function has(Set storage q, bytes32 o) internal view returns (bool) {
        return q.index[o] > 0 && q.index[o] <= q.data.length;
    }
```

POC

Put the file in test/POC.t.sol

https://gist.github.com/thangtranth/685dd8fa7faae141cdd2b1d0061b16f5

Recommendations

The `remove` function in `AsSequentialSet.sol` should be modified to reset the `index` of the removed account

```
function remove(Set storage q, bytes32 o) internal {
        uint32 i = q.index[o];
+       q.index[o] = 0;
        require(i > 0, "Element not found");
        removeAt(q, i - 1);
    }
```

[H-02] Flash loan wrong balance check

Severity

Impact: Medium, because the flash loan functionality is affected

Likelihood: High, because it will revert every time a flash loan is used

Description

`balanceBefore` is recorded before the flash loan amount being transferred. As a result, in line 714, `balanceAfter` need to be more than needed. Users have to pay extra with the same amount to use a flash loan.

```
File: src\abstract\As4626.sol
696:     function flashLoanSimple() external nonReentrant {

705:         uint256 fee = exemptionList[msg.sender] ? 0 : amount.bp
  (fees.flash);
706:         uint256 toRepay = amount + fee;
707:
708:         uint256 balanceBefore = asset.balanceOf(address(this));
709:         totalLent += amount;
710:
711:         asset.safeTransferFrom(address(this), address(receiver), amount);
712:         receiver.executeOperation(address
  (asset), amount, fee, msg.sender, params);
713:
714:         if ((asset.balanceOf(address(this)) - balanceBefore) < toRepay)
715:             revert FlashLoanDefault(msg.sender, amount);

718:     }
```

Recommendations

The `toRepay` variable could be dropped, or record the `balanceBefore` after `safeTransferFrom`:

```
File: src\abstract\As4626.sol
696:     function flashLoanSimple() external nonReentrant {

705:         uint256 fee = exemptionList[msg.sender] ? 0 : amount.bp
  (fees.flash);
- 706:         uint256 toRepay = amount + fee;
707:
708:         uint256 balanceBefore = asset.balanceOf(address(this));
709:         totalLent += amount;
710:
711:         asset.safeTransferFrom(address(this), address(receiver), amount);
712:         receiver.executeOperation(address
  (asset), amount, fee, msg.sender, params);
713:
- 714:         if ((asset.balanceOf(address(this)) - balanceBefore) < toRepay)
+ 714:         if ((asset.balanceOf(address(this)) - balanceBefore) < fee)
715:           revert FlashLoanDefault(msg.sender, amount);

718:     }
```

[H-03] Flash loan not working due to transferFrom Issue

Severity

Impact: Medium - The flash loan functionality is non-operational, but there's no risk of fund loss.

Likelihood: High - The flash loan function consistently fails to execute as intended.

Description

The issue arises when users attempt to use the `flashLoanSimple` function:

```
function flashLoanSimple(
            IFlashLoanReceiver receiver,
            uint256 amount,
            bytes calldata params
        ) external nonReentrant {

            uint256 available = availableBorrowable();
            if (amount > available || amount > maxLoan) revert AmountTooHigh
              (amount);

            uint256 fee = exemptionList[msg.sender] ? 0 : amount.bp(fees.flash);
            uint256 toRepay = amount + fee;

            uint256 balanceBefore = asset.balanceOf(address(this));
            totalLent += amount;

@>          asset.safeTransferFrom(address(this), address(receiver), amount);
            receiver.executeOperation(address
              (asset), amount, fee, msg.sender, params);

            if ((asset.balanceOf(address(this)) - balanceBefore) < toRepay)
                revert FlashLoanDefault(msg.sender, amount);

            emit FlashLoan(msg.sender, amount, fee);
        }
```

To transfer the fund to users, it uses `asset.safeTransferFrom(address(this), address(receiver), amount);`

This line intends to transfer funds to the user. However, it fails because `safeTransferFrom` requires the contract to have a sufficient `allowance` to "spend" on behalf of itself. In the context of ERC20 tokens like USDC, the transferFrom function includes a crucial check: `value <= allowed[from][msg.sender]`

However, because the contract has not yet approved itself, leading to a situation where the allowance remains at zero, and hence the `transferFrom` call reverts.

```
function transferFrom(
        address from,
        address to,
        uint256 value
    )
        external
        override
        whenNotPaused
        notBlacklisted(msg.sender)
        notBlacklisted(from)
        notBlacklisted(to)
        returns (bool)
    {
        require(
            value <= allowed[from][msg.sender],
            "ERC20: transfer amount exceeds allowance"
        );
        _transfer(from, to, value);
        allowed[from][msg.sender] = allowed[from][msg.sender].sub(value);
        return true;
    }
```

USDC - FiatTokenV1.sol:
https://arbiscan.io/address/0xaf88d065e77c8cc2239327c5edb3a432268e5831

Using `transfer` does not require additional approval.

Recommendations

Replacing the `safeTransferFrom` function with `safeTransfer`:

```
function flashLoanSimple(
            IFlashLoanReceiver receiver,
            uint256 amount,
            bytes calldata params
        ) external nonReentrant {

            uint256 available = availableBorrowable();
            if (amount > available || amount > maxLoan) revert AmountTooHigh
              (amount);

            uint256 fee = exemptionList[msg.sender] ? 0 : amount.bp(fees.flash);
            uint256 toRepay = amount + fee;

            uint256 balanceBefore = asset.balanceOf(address(this));
            totalLent += amount;

-           asset.safeTransferFrom(address(this), address(receiver), amount);
+           asset.safeTransfer(address(receiver), amount);
            receiver.executeOperation(address
              (asset), amount, fee, msg.sender, params);

            if ((asset.balanceOf(address(this)) - balanceBefore) < toRepay)
                revert FlashLoanDefault(msg.sender, amount);

            emit FlashLoan(msg.sender, amount, fee);
        }
```

[H-04] `withdraw()` worst price will distort `sharePrice()`

Severity

Impact: High, because inaccurate price will cause long term value leaking, and can not be fixed

Likelihood: Medium, because every time `withdraw()` with existing request will bring in error into the system

Description

The worst price is used in `_withdraw()`, which means the actual withdraw price is different from `last.sharePrice`.

```
File: src\abstract\As4626.sol
149:     function _withdraw() internal nonReentrant returns (uint256) {
159:         last.sharePrice = sharePrice();

161:         uint256 price = (claimable >= _shares)
162:             ? AsMaths.min
//(last.sharePrice, request.sharePrice) // worst of if pre-existing request
163:             : last.sharePrice; // current price
```

`last.sharePrice` is only updated in the following cases, none of them take the real withdraw price into consideration.``

```
File: src\abstract\As4626.sol
84:     function _deposit() internal nonReentrant returns (uint256) {
93:         last.sharePrice = sharePrice();

149:     function _withdraw() internal nonReentrant returns (uint256) {
159:         last.sharePrice = sharePrice();

512:     function requestRedeem() public nonReentrant {
523:         last.sharePrice = sharePrice();

577:     function cancelRedeemRequest() external nonReentrant {
590:         last.sharePrice = sharePrice();
```

Let's look at how `sharePrice()` is calculated: First it calls `totalAccountedAssets()`, and in `totalAccountedAssets()` the `last.sharePrice` is directly used, seems the assumption is: `last.sharePrice` remain constant with each `deposit()/withdraw()`. However it does not hold since the worst price usage.

```
File: src\abstract\As4626Abstract.sol
175:     function sharePrice() public view virtual returns (uint256) {
176:         uint256 supply = totalAccountedSupply();
177:         return
178:             supply == 0
179:                 ? weiPerShare
180:  @>>>>          : totalAccountedAssets().mulDiv( // eg. e6
181:                     weiPerShare ** 2, // 1e8*2
182:                     supply * weiPerAsset
183:                 ); // eg. (1e6+1e8+1e8)-(1e8+1e6)
184:     }

154:     function totalAccountedAssets() public view returns (uint256) {
155:         return
156:             totalAssets() -
157:             req.totalClaimableRedemption.mulDiv(
158:  @>>>>          last.sharePrice * weiPerAsset,
159:                 weiPerShare ** 2
160:             ); // eg. (1e8+1e8+1e6)-(1e8+1e8) = 1e6
161:     }
```

To summarize, when `request.sharePrice < sharePrice()`, the withdrawal will processed at a lower price than current, in which case the discrepancy would incur an effective `sharePrice` increase that would not be factored-in `last.sharePrice`.

Let's see some number example:

- at first, total share is 100, total asset 500, sharePrice is 5.
- some user with 20 shares request withdraw at price 5. totalClaimableRedemption is 40 (total 20% on hold).
- after a long time, total asset becomes 1000, sharePrice is 10, and liquidity is enough.
- Alice calls withdraw() at price 5 (due to worst price in _withdraw()), total asset becomes 80 (60 + 20 in totalClaimableRedemption).

Now total asset is 1000 - 20 * 5 = 900. But last.sharePrice remains 10, due to As4626.sol#159 (last.sharePrice = sharePrice()). The actual price should be 900 / 80 = 11.25.

The next time totalAccountedAssets() will return 900 - 20 * 10 = 700, sharePrice() will return 700 / 60 = 11.67, around 3.7% difference (11.67~11.25).

So if some user tries to deposit a bit amount, 3.7% slippage will be the loss.

```
File: src\abstract\As4626Abstract.sol
154:     function totalAccountedAssets() public view returns (uint256) {
155:         return
156:             totalAssets() -
157:             req.totalClaimableRedemption.mulDiv(
158:                 last.sharePrice * weiPerAsset,
159:                 weiPerShare ** 2
160:             ); // eg. (1e8+1e8+1e6)-(1e8+1e8) = 1e6
161:     }
```

I believe in normal situations, if the last.sharePrice and request.sharePrice are relatively close, the sharePrice will only diff slightly.

Recommendations

One possible mitigation is to update the `last.sharePrice` after `deposit()/withdraw()`.

[H-05] Fee calculation mismatch in `mint`, `deposit`, `redeem` and `withdraw`

Severity

Impact: Medium, fee will be a little higher/lower

Likelihood: High, because it happens in every call to mint and deposit functions

Description

When users calls `mint(shares)` code calls `_deposit(previewMint(_shares), _shares` and `previewMint(shares) = convertToAssets(shares).addBp()`. When users calls `deposit(amount)` code calls `_deposit(_amount, previewDeposit(_amount)` and `previewDeposit(amount) = convertToShares(amount).subBp`.

Let's assume that price is 1:1 and fee is 10% and check the both case:

1. If user wants to mint 100 share then he would call `mint(100)` and code would calculate `amount = previewMint(100) = convertToAssets(100).addBp(10%) = 110`. So in the end user would pay 110 asset and receive 100 shares and 10 asset will be fee.

2. If users wants to deposit 110 asset then he would call `deposit(110)` and code would calculate `share = previewDeposit(110) = convertToShare(110).subBp(10%) = 99`. So in the end user would pay 11 asset and receive 99 share and 11 asset will be fee.

As you can see the `deposit()` call overcharge the user. The reason is that code calculates fee based on user-specified amount by using `subBp()` but user-specified amount is supposed to be `amount + fee` so the calculation for fee should be `.... * base / (base +fee)`.

When users call `redeem(shares)` code calls `_withdraw(previewRedeem(_shares), _shares)` and `previewRdeem(shares) = convertToAssets(_shares).subBp()`.

When users call `withdraw()` code calls `_withdraw(_amount, previewWithdraw(_amount))` and `previewWithdraw(_amount) = convertToShares(_assets).addBp()`

Let's assume that asset to share price is 1:1 and fee is 10% and check both case:

1. If user wants to redeem 110 shares then he would call `redeem(110)` and code would calculate `amount = previewRdeem(110) = convertToAssets(110).subBp(10%) = 99`. So in the end user would burn 110 shares and receive 99 asset and 11 asset will be fee.
2. If user wants to withdraw 100 asset then he would call `withdraw(100)` and code would calculates `shares = previewWithdraw(100) = convertToShares(100).addBp(10%) = 110`. So in the end userwould burn 110 shares and receive 100 asset and 10 asset will be fee.

So as you can see `redeem()` overcharges users. The reason is that code calculates fee based on user provided share with `subBp()` but the provided amount is total amount (burnAmount + fee) and calculation should be `..... * base / (base + fee)`

Recommendations

Calculate the fee for `deposit()` with `convertToShare(amount) * base / (base +fee)`. Calculate the fee for `previewRedeem()` with `convertToAssets(shares) * base / (base + fee)`

[H-06] Fee target mismatch in `deposit`, `mint`, `withdraw`, `redeem` and `preview***` methods

Severity

Impact: High, because it can cause double spending and disturb the pool calculations

Likelihood: Medium, because exception recipients are set by admin for common addresses

Description

When users want to deposit their tokens, code calculates fee in `previewDeposit()` and `previewMint()` and add/subtract it from the amount/share. As you can see code checks entry fee based on `msg.sender`:

```
function previewDeposit(
        uint256 _amount
    ) public view returns (uint256 shares) {
        return convertToShares(_amount).subBp
          (exemptionList[msg.sender] ? 0 : fees.entry);
    }

    function previewMint(uint256 _shares) public view returns (uint256) {
        return convertToAssets(_shares).addBp
          (exemptionList[msg.sender] ? 0 : fees.entry);
    }
```

In `_deposit()`, code wants to account the fee and keep track of it, it perform this action:

```
if (!exemptionList[_receiver])
            claimableAssetFees += _amount.revBp(fees.entry);
```

As you can see it uses `_receiver` variable which is controllable by caller.

So attacker with two addresses: RECV1 address and has set as `exemptionList[]` can ADDERSS1 which doesn't set as `exemptionList[]` can call mint and deposit with ADDRESS1 and set recipient as RECV1 . In preview functions code doesn't add the fee for user deposit(or subtract it from shares) and code would assume user didn't pay any fee, but in the `_deposit()`

function code would check RECV1 address and would add calculated fee to accumulated fee. So in the end user didn't paid the fee but code added fee and double spending would happen.

Attacker can use another scenarios to perform this issue too. (code calculates fee and caller pays it but code doesn't add it to `claimableAssetFees`)

When users want to withdraw their tokens, Code charges fee and it's done in preview functions by adding/subtracting fee from amount/share. As you can see code checks exit fee based on `exemptionList[]` and uses `msg.sender` as target:

```
function previewWithdraw(uint256 _assets) public view returns (uint256) {
        return convertToShares(_assets).addBp
          (exemptionList[msg.sender] ? 0 : fees.exit);
    }

    function previewRedeem(uint256 _shares) public view returns (uint256) {
        return convertToAssets(_shares).subBp
          (exemptionList[msg.sender] ? 0 : fees.exit);
    }
```

But in `_withdraw()` function when code wants to calculates fee and accumulated it, it uses `_owner`:

```
if (!exemptionList[_owner])
            claimableAssetFees += _amount.revBp(fees.exit);
```

`owner` can be different that caller(`msg.sender`) and code checks that caller have approval over the `owner`'s funds. So attacker can exploit this with two of his address: OWNER1 which has set as `exemptionList[]` and OPERATOR1 which doesn't set as `exemptionList[]`. If attacker give approval of OWNER1 tokens to OPERATOR1 and calls `withdraw(OWNER1)` with OPERATOR1 address then double spend would happen. While code returns funds fully with no charged fee it would also add fee to `claimableAssetFees`.

This can be exploited in other scenarios too. In general this inconsistency would cause accountant errors.

Recommendations

Calculate the fee based on `msg.sender` in the `_deposit()` function. In `_withdraw()` function calculate fee based on `msg.sender` and finally fix the `preview` functions.

[H-07] Accounting issues after underlying asset change

Severity

Impact: High, because the accounting would go wrong for multiple scenarios

Likelihood: Medium, because it would happen when admin calls `changeAsset()`

Description

In general updating underlying asset is very risky move in a pool. All the cached prices will be wrong.

In the current code we have two cached prices(that I know of): In requestRedeem() code caches pool prices for requests. Code use it later in the withdraw and cancel request. (the price impact withdraw price and also burning tokens in cancel requests)

In calculating fee, code caches pool price and use it to calculate fee later.

(there may be other places the pool price is cached)

——— Another place that is asset amount is cached is `claimableAssetFees`. `updateAsset()` calls the `_collectFees()` to handle the `claimableAssetFees` and set it to zero but because of this line in the `_collectFees()` `If (profit == 0) return;` `claimableAssetFees` (which shows amount in old asset) could remain non-zero after asset update.

Recommendations

Reset the cached prices after the asset change.

8.3. Medium Findings

[M-01] Redeem function active when vault is paused

Severity

Impact: High - Allows unauthorized withdrawal of assets during critical situations when the vault is paused.

Likelihood: Low - This issue occurs only when the vault is in a paused state.

Description

The vault's deposit, mint, and withdraw functionalities are halted when it is paused. This is implemented through the `whenNotPaused` modifier in the following functions:

```
function deposit(
        uint256 _amount,
        address _receiver
    ) public whenNotPaused returns (uint256 shares) {

    function safeDeposit(
        uint256 _amount,
        uint256 _minShareAmount,
        address _receiver
    ) public whenNotPaused returns (uint256 shares) {

    function withdraw(
        uint256 _amount,
        address _receiver,
        address _owner
    ) external whenNotPaused returns (uint256) {

    function safeWithdraw(
        uint256 _amount,
        uint256 _minAmount,
        address _receiver,
        address _owner
    ) public whenNotPaused returns (uint256 amount) {
```

However, the redeem function is not pausable because the `whenNotPaused` modifier is not applied . This absence allows users to withdraw assets from the vaul when they should not.

```
function redeem(
        uint256 _shares,
        address _receiver,
        address _owner
    ) external returns (uint256 assets) {

    function safeRedeem(
        uint256 _shares,
        uint256 _minAmountOut,
        address _receiver,
        address _owner
    ) external returns (uint256 assets) {
```

Recommendations

Adding the whenNotPaused modifier to both the redeem and safeRedeem functions.

```
function redeem(
        uint256 _shares,
        address _receiver,
        address _owner
-   ) external returns (uint256 assets) {
+   ) external whenNotPaused returns (uint256 assets) {
        return _withdraw(previewRedeem(_shares), _shares, _receiver, _owner);
    }

    function safeRedeem(
        uint256 _shares,
        uint256 _minAmountOut,
        address _receiver,
        address _owner
-    ) external returns (uint256 assets) {
+    ) external whenNotPaused returns (uint256 assets) {
        assets = _withdraw(
            previewRedeem(_shares),
            _shares, // _shares
            _receiver, // _receiver
            _owner // _owner
        );
        if (assets < _minAmountOut) revert AmountTooLow(assets);
    }
```

[M-02] Wrong rounding direction in previewWithdraw()

Severity

Impact: Low, because revert in mint() and violates EIP4626

Likelihood: High, because division happens in each tx

Description

According to the EIP4626 `previewWithdraw` should round up when performing division:

> Finally, EIP-4626 Vault implementers should be aware of the need for specific, opposing rounding directions across the different mutable and view methods, as it is considered most secure to favor the Vault itself during calculations over its users:
>
> If (1) it's calculating how many shares to issue to a user for a certain amount of the underlying tokens they provide or (2) it's determining the amount of the underlying tokens to transfer to them for returning a certain amount of shares, it should round down.
>
> If (1) it's calculating the amount of shares a user has to supply to receive a given amount of the underlying tokens or (2) it's calculating the amount of underlying tokens a user has to provide to receive a certain amount of shares, it should round up.

But in current implementation code rounds down and favors the caller instead of the contract.

Function `withdraw()` uses `previewWithdraw()` to calculate shares and calls `_withdraw()`, as there is a check for price in `_withdraw()` to make sure user received price isn't better than current price, so that check will fail and cause revert when rounding errors happens. (calculated `_shares` will be smaller and the right side of the condition will be smaller)

```
// amount/shares cannot be higher than the share price
        //(dictated by the inline convertToAssets below)
        if (_amount >= _shares.mulDiv(price * weiPerAsset, weiPerShare ** 2))
            revert AmountTooHigh(_amount);
```

Recommendation

Change `previewWithdraw` so that it rounds up when calculating shares.

[M-03] Wrong rounding direction in previewMint()

Severity

Impact: Low, because it violates EIP and also cause revert in mint()

Likelihood: High, because division happens in every mint() call

Description

According to the EIP4626 function `previewMint()` should round up when calculating assets:

> Finally, EIP-4626 Vault implementers should be aware of the need for specific, opposing rounding directions across the different mutable and view methods, as it is considered most secure to favor the Vault itself during calculations over its users:
>
> If (1) it's calculating how many shares to issue to a user for a certain amount of the underlying tokens they provide or (2) it's determining the amount of the underlying tokens to transfer to them for returning a certain amount of shares, it should round down.
>
> If (1) it's calculating the amount of shares a user has to supply to receive a given amount of the underlying tokens or (2) it's calculating the amount of underlying tokens a user has to provide to receive a certain amount of shares, it should round up.

in current implementation code rounds down. this will cause calculations to be in favor of the caller instead of the contract.

in function `mint()` code uses `previewMint()` and calls `_deposit()`, as `previewMint()` would calculate smaller amount for `_amount` so the check inside the `_deposit()` that makes sure user don't receive better price than current price would fail and call would revert: (`_amount` would be lower a little and cause right side of the condition to be smaller)

```
if (_shares > _amount.mulDiv
          (weiPerShare ** 2, last.sharePrice * weiPerAsset))
            revert AmountTooHigh(_amount);
```

Recommendations

Change `previewMint` so that it rounds up when calculating shares.

[M-04] Wrong usage of revBp() in deposit() in fee calculation

Severity

Impact: Low, because wrong accounting of fee

Likelihood: High, because it will happen in each call to deposit/mint

Description

In `_deposit()` function code calculates fee like this:

```
// slice the fee from the amount (gas optimized)
        if (!exemptionList[_receiver])
            claimableAssetFees += _amount.revBp(fees.entry);
```

And the `revBp()` logic is:

```
function revBp(
        uint256 amount,
        uint256 basisPoints
    ) internal pure returns (uint256) {
        return mulDiv(amount, basisPoints, BP_BASIS - basisPoints);
    }
```

As the `amount` in the deposit is not sliced and it is `deposit + fee` so the calculation for fee is wrong.

Recommendations

Fee calculation should be:

```
claimableAssetFees  += _amount * fees.entry /
                (BP_BASIS  + fees.entry)
```

[M-05] Using deprecated function in Chainlink

Severity

Impact: High - Using stale prices leads to inaccurate calculations of total asset values and share prices.

Likelihood: Low - The return price can be wrong or stale without validating

Description

To convert from USD to input token amount, `StrategyV5Chainlink` uses `IChainlinkAggregatorV3.latestAnswer`. However the function `latestAnswer` is deprecated by Chainlink. This deprecated function usage is also observed in other libraries, such as `ChainlinkUtils`.

```
function _usdToInput(uint256 _amount, uint8 _index) internal view returns
  (uint256) {
    return _amount.mulDiv(10**uint256
      (inputFeedDecimals[_index]) * inputDecimals[_index],
        uint256(inputPriceFeeds[_index].latestAnswer()) * 1e6); // eg.
        //(1e6+1e8+1e6)-(1e8+1e6) = 1e6
}
```

For reference: https://docs.chain.link/data-feeds/api-reference#latestanswer

`IChainlinkAggregatorV3.latestRoundData` should be used instead.

Recommendations

Update the function to use `latestRoundData` from Chainlink. This method provides comprehensive data about the latest price round, including the timestamp, ensuring the price's freshness and relevance.

Example implementation:

```
uint256 private constant GRACE_PERIOD_TIME = 3600; // how long till we consider
// the price as stale

function getChainlinkPrice (AggregatorV2V3Interface feed) internal {
    (
      uint80roundId,
      int256price,
      uintstartedAt,
      uintupdatedAt,
      uint80answeredInRound
    ) = feed.latestRoundData(
    require(price > 0, "invalid price");
    require(block.timestamp <= updatedAt + GRACE_PERIOD_TIME, "Stale price");
    return price;
}
```

When deploying on Arbitrum, include a check to verify the status of the Arbitrum Sequencer, as this can impact the reliability of the price feeds.

Example: https://docs.chain.link/data-feeds/l2-sequencer-feeds#example-code

[M-06] Fee on Transfer Token Will Break accounting

Severity

Impact: High, because the accounting will be incorrect, and the sharePrice will be affected

Likelihood: Low, because fee on transfer token is not commonly used

Description

`mint()/deposit()` is using `amount` for transfering and accounting. But fee on transfer token could break the accounting, since the actual token received will be less than amount. As a result, `sharePrice` will have some small error each time.

```
File: src\abstract\As4626.sol
69:     function mint(
70:         uint256 _shares,
71:         address _receiver
72:     ) public returns (uint256 assets) {
73:         return _deposit(previewMint(_shares), _shares, _receiver);
74:     }

117:     function deposit(
118:         uint256 _amount,
119:         address _receiver
120:     ) public whenNotPaused returns (uint256 shares) {
121:         return _deposit(_amount, previewDeposit(_amount), _receiver);
122:     }

84:     function _deposit(
85:         uint256 _amount,
86:         uint256 _shares,
87:         address _receiver
88:     ) internal nonReentrant returns (uint256) {

98:         asset.safeTransferFrom(msg.sender, address(this), _amount);

105:         _mint(_receiver, _shares);
```

USDT potentially could turn on fee on transfer feature, but not yet.

Recommendations

Use before and after balance to accurately reflect the true amount received, and update share price accordingly.

[M-07] Using stale price in Pyth Network

Severity

Impact: High - Using stale prices leads to inaccurate calculations of total asset values and share prices.

Likelihood: Low - Price can be stale frequently if there is no update

Description

The `StrategyV5Pyth` uses `pyth.getPriceUnsafe` for obtaining Pyth oracle price feeds to calculate the asset/input exchange rate.

```
function assetExchangeRate(uint8 inputId) public view returns (uint256) {
        if (inputPythIds[inputId] == assetPythId)
            return weiPerShare; // == weiPerUnit of asset == 1:1
        PythStructs.Price memory inputPrice = pyth.getPriceUnsafe
          (inputPythIds[inputId]);
        PythStructs.Price memory assetPrice = pyth.getPriceUnsafe(assetPythId);
        ...
    }
```

However, from the Pyth documents, using the getPriceUnsafe can return stale price if the price is not updated.

```
/// @notice Returns the price of a price feed without any sanity checks.
    /// @dev This function returns the most recent price update in this contract
    // without any recency checks.
    /// This function is unsafe as the returned price update may be arbitrarily
    // far in the past.
    ///
    /// Users of this function should check the `publishTime` in the price to
    // ensure that the returned price is
    /// sufficiently recent for their application. If you are considering using
    // this function, it may be
    /// safer / easier to use either `getPrice` or `getPriceNoOlderThan`.
    /// @return price - please read the documentation of PythStructs.Price to
    // understand how to use this safely.
    function getPriceUnsafe(
        bytes32 id
    ) external view returns (PythStructs.Price memory price);
```

The `assetExchangeRate` function doesn't verify `Price.publishTime`, potentially leading to outdated exchange rates, incorrect investment calculations, and distorted total asset values.

Recommendations

Using `pyth.updatePriceFeeds` for updating prices, followed by `pyth.getPrice` for retrieval. Following the example in: https://github.com/pyth-network/pyth-sdk-solidity/blob/main/README.md#example-usage

[M-08] `ERC20::approve` will revert for some non-standard tokens like USDT

Severity

Impact: Medium, because functionality won't work

Likelihood: Medium, because USDT is a common token

Description

Code uses the `approve` method to set allowance for ERC20 tokens in `setSwapperAllowance`. This will cause revert if the target ERC20 was a non-standard token that has different function signature for `approve()` function. Tokens like USDT will cause revert for this function, so they can't be used as reward token, input token and underlying asset.

```
function setSwapperAllowance(uint256 _amount) public onlyAdmin {
        address swapperAddress = address(swapper);

        for (uint256 i = 0; i < rewardLength; i++) {
            if (rewardTokens[i] == address(0)) break;
            IERC20Metadata(rewardTokens[i]).approve(swapperAddress, _amount);
        }
        for (uint256 i = 0; i < inputLength; i++) {
            if (address(inputs[i]) == address(0)) break;
            inputs[i].approve(swapperAddress, _amount);
        }
        asset.approve(swapperAddress, _amount);
    }
```

Recommendations

Use `SafeERC20`'s `forceApprove` method instead to support all the ERC20 tokens.

8.4. Low Findings

[L-01] Inconsistency in access control for setInputs

In both `StrategyV5Chainlink` and `StrategyV5Pyth`, the `setInputs` function is restricted to the `admin` role:

```
function setInputs(
      address[]calldata_inputs,
      uint16[]calldata_weights,
      address[]calldata_priceFeeds
    ) external onlyAdmin {
        ...
    }

    function setInputs(
      address[]calldata_inputs,
      uint16[]calldata_weights,
      bytes32[]calldata_pythIds
    ) external onlyAdmin {
        ...
    }
```

Contrastingly, in the `StrategyV5Agent` (the parent contract), `setInputs` is designed to be accessible by the `manager` role:

```
function setInputs(
        address[] calldata _inputs,
        uint16[] calldata _weights
    ) public onlyManager {
        ...
    }
```

If an account holds only the admin or manager role, they cannot execute the `setInputs` function in all strategies. And it may lead to confusion.

To resolve this, make sure to be consistent in the access control of the `setInputs` method.

[L-02] Storage slot collision due to adding rescueRequests in StrategyV5Agent

The storage layout of a strategy is defined by `StrategyV5Abstract` and `As4626Abstract`. The strategy contract inherents these two contracts for its storage variable definition.

The StrategyV5Chainlink inherents storage of 2 above contracts and adds 4 other storage variables:

```
IChainlinkAggregatorV3 internal assetPriceFeed; // Aggregator contract of the
// asset asset
IChainlinkAggregatorV3[8] internal inputPriceFeeds; // Aggregator contract of
// the inputs
uint8 internal assetFeedDecimals; // Decimals of the asset asset
uint8[8] internal inputFeedDecimals; // Decimals of the input asset
```

Inspecting the storage layout of `StrategyV5Chainlink`, at slot 55 it stores the `inputPriceFeeds` state variable.

However, in the implementation contract `StrategyV5Agent`. It adds one storage slot `mapping(address => RescueRequest) private rescueRequests`. It also uses the slot `55` and collides with `inputPriceFeeds` in `StrategyV5Chainlink`.

In this case, due to the `rescueRequests` is a mapping, the real data of the mapping is stored in `slot = keccak256([key, mappingSlot])`, so no data is corrupted. However, it may introduce some unpredictability if other data type is added to the `StrategyV5Agent` in latter version and should be avoid. It's crucial to maintain a clear and collision-free storage layout to ensure contract stability and predictability.