Astrolab DAO
Security Review

Version 2.0

Jan 31, 2024

Conducted by:

MaslarovK and Solthodox, Independent Security Researchers

Table of Contents

1 About Solthodox

Solthodox is a smart contract developer and independent security researcher experienced in Solidity smart contract development and transitioning to security. With +1 year of experience in the development side, he has been joining security contests in the last few months. He also serves as a smart contract developer at Unlockd Finance, where he has been involved in building defi yield farming strategies to maximze the APY of it's users.

2 About MaslarovK

MaslarovK is an independent security researcher from Bulgaria with 3 years of experience in Web2 development. His curiosity and love for decentralisation and transparency made him transition to Web3. He has secured various protocols through public contests and private audits.

3 Disclaimer

Audits are a time, resource, and expertise bound effort where trained experts evaluate smart contracts using a combination of automated and manual techniques to identify as many vulnerabilities as possible. Audits can show the presence of vulnerabilities **but not their absence**.

4 Risk classification

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

4.1 Impact

- **High** - leads to a significant loss of assets in the protocol or significantly harms a group of users.
- **Medium** - only a small amount of funds can be lost or a functionality of the protocol is affected.
- **Low** - any kind of unexpected behaviour that's not so critical.

4.2 Likelihood

- **High** - direct attack vector; the cost is relatively low to the amount of funds that can be lost.
- **Medium** - only conditionally incentivized attack vector, but still relatively likely.
- **Low** - too many or too unlikely assumptions; provides little or no incentive.

4.3 Actions required by severity level

- **Critical** - client **must** fix the issue.
- **High** - client **must** fix the issue.
- **Medium** - client **should** fix the issue.
- **Low** - client **could** fix the issue.

5 Executive summary

Overview

Project Name	AstroLabs Strats
Repository	https://github.com/AstrolabDAO/strats
Commit hash	82ad070202c210c8b12133afbdb63c68c9f42918
Resolution	89df2a413336eca597cbc8aa9abc59424df2ebd0
Documentation	https://docs.astrolab.fi/introduction/overview.html
Methods	Manual review & testing

Scope

src/abstract/AsAccessControl.sol
src/abstract/AsRescuable.sol
src/abstract/StrategyV5Abstract.sol
src/abstract/As4626.sol
src/abstract/StrategyV5.sol
src/abstract/StrategyV5Chainlink.sol
src/abstract/AsManageable.sol
src/abstract/StrategyV5Agent.sol
src/abstract/StrategyV5Pyth.sol
src/abstract/As4626Abstract.sol
src/libs/AsAccounting.sol

Issues Found

Critical risk	1
High risk	3
Medium risk	7
Low risk	10
Informational	11

6 Findings

6.1 Critical risk

6.1.1 Incorrect decimal scalation will result in very wrong prices

Severity: *Critical risk*

Context: StrategyV5Chainlink.sol#L124C4-L138C6

Description: Within the `StrategyV5Chainlink` contract, specifically in the view functions `_usdToInput` and `_inputToUsd`, there is a miscalculation in decimal scaling during token-to-USD and USD-to-token conversions. The error is evident in the `_usdToInput` function, where the multiplication factor lacks correct decimal scaling, potentially resulting in inaccurate calculations.

But the decimal scalation is not correct, for instance in `_assetToInput`:

```
function _usdToInput(uint256 _amount, uint8 _index) internal view returns (uint256)
    {
    // Incorrect decimal scaling: amount * (10 ** feedDecimals * inputDecimals)
    return _amount.mulDiv(10**uint256(inputFeedDecimals[_index]) * inputDecimals[
        _index],
    //...
```

Proof of Concept:

```
decimalsA = 5
decimalsB = 8
amount = 100

// Incorrect scaling:
escalated = amount * (10 ** decimalsA * decimalsB) = amount * 80000

// Correct scaling:
escalated = amount * (10 ** (decimalsA + decimalsB)) = amount * 10000000000000
```

Recommendation: The mathematical error can be rectified by adjusting the scaling in both functions. Here's the corrected code:

```
function _usdToInput(uint256 _amount, uint8 _index) internal view returns (uint256)
    {
    // Corrected scaling: amount * (10 ** (feedDecimals + inputDecimals))
    return _amount.mulDiv(10**(uint256(inputFeedDecimals[_index]) + inputDecimals[
        _index]),
        uint256(inputPriceFeeds[_index].latestAnswer()) * 1e6);
}

function _inputToUsd(uint256 _amount, uint8 _index) internal view returns (uint256)
    {
    // Corrected scaling: amount * (10 ** (feedDecimals + inputDecimals))
    return _amount.mulDiv(uint256(inputPriceFeeds[_index].latestAnswer()) * 1e6,
        10**(uint256(inputFeedDecimals[_index]) + inputDecimals[_index]));
}
```

Resolution: Resolved

6.2 High risk

6.2.1 Inconsistent and incorrect entry and exit fees calculation in As4626

Severity: *High risk*

Description: There are several errors regarding the calculations of the entry and exit fees in As4626 .sol and also it's done inconsistantly, since sometimes they are calculated from shares and others from assets.

Proof of Concept:

There are several errors regarding the fees calculations:

1. In the _deposit and function in As4626 the fees are always sliced from the amount, while previewDeposit always substracts them from the shares and previewMint from the assets. The same goes for _withdraw and previewWithdraw and previewRedeem.

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

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

To slice the fees it calculates the reverse fee, which means that the contract assumes that the fees have already been substracted in the preview, but that only happens in previewMint. The inconsistency in the fee calculation method leads to different real fee percentages depending on the deposit method used.

```
// this is only correct when using "mint()"
if (!exemptionList[_receiver])
            claimableAssetFees += _amount.revBp(fees.entry);

// previewMint returns amount.subBp(exemptionList[msg.sender] ? 0 : fees.entry);
// previewDeposit returns amount
```

2. previewMint and previewWithdraw use addBp instead of revSubBp. previewMint for instance should have the inverse effect of previewDeposit, but it doesn't :

```
// lets say shares/ assets are 1:1
// fee is 20%

// if we we do the previewMint :
convertToAssets(100) = 100 // convert to assets(1:1 relation)
100 .addBp(20%) = 120 // add 20% fees with addBp


// if we do previewDeposit(120) we should get 100
```

```
convertToShares(120)
120.subBp(20%) = 96
and after substracting the fees : 120 * 20%
you are left with 96
not 100
```

3. The fees for the collector are saved into the contract as assets, and minted to him as shares when he collects the fees meaning that the amount of shares to be minted depends on the moment he collects them. Additionally, accounting them as shares is more standard and will acrrue interest for the collector since the very moment of the fees being accounted.

Recommendation: Always calculate fees from shares, and use `revSubBp` instead of `addBp` in the previews:

```
    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.revAddBp(exemptionList[msg.sender] ? 0 : fees
            .entry));
    }
```

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

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

If when accruing fees we actually mint them, the share prices will be automatically adjusted accordingly.

```
// _deposit
if (!exemptionList[_receiver])
            uint256 claimableShareFees = _shares.revBp(fees.entry);
            _mint(address(this),claimableShareFees);
```

```
// _withdraw
if (!exemptionList[_owner])
            uint256 claimableSharesFees = _shares_.revBp(fees.exit);
            _mint(address(this),claimableShareFees);
```

In the `collectFees` function just transfer the previously minted share fees to the collector:

```
function _collectFees() internal nonReentrant {

        if (feeCollector == address(0))
            revert AddressZero();
        (uint256 assets, uint256 price, uint256 profit, uint256 feesAmount) =
            AsAccounting.computeFees(IAs4626(address(this)));

        if (profit == 0) return;
        uint256 toMint = convertToShares(feesAmount)
        emit FeeCollection(
            feeCollector,
            assets,
            price,
            profit, // basis AsMaths.BP_BASIS**2
            feesAmount,
            toMint
        );
        // mint managment + performance fees
        _mint(feeCollector, toMint);
        // transfer share fees
        transfer(feeCollector, claimableSharesFees);
        last.feeCollection = uint64(block.timestamp);
        last.accountedAssets = assets;
        last.accountedSharePrice = price;
        last.accountedProfit = profit;
        last.accountedSupply = totalSupply();

        // set it to make clear all have been transferred already
        claimableAssetFees = 0;
}
```

```
//As4626Abstract
function availableClaimable() internal view returns (uint256) {
        return
            asset.balanceOf(address(this)).subMax0(
                // we can remove this line : claimableAssetFees +
                    AsAccounting.unrealizedProfits(
                        last.harvest,
                        expectedProfits,
                        profitCooldown
                    )
            );
    }
```

Resolution: Resolved

6.2.2 Malicious users can create redeem requests on behalf of any `operator`

Severity: *High risk*

Context: As4626.sol#L512

Description: The `requestRedeem` function in `As4626` allows to set any `operator` without any constraints and create/modify redeem requests for them.

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

        // request = storage pointer of the request to modify
        // operator can really be arbitrary
        Erc7540Request storage request = req.byOperator[operator];
        if (request.operator != operator) request.operator = operator;
    //..
}
```

Proof of Concept:

This can harm the `operator` in several ways:

1. Create a redeem request in a moment where the share price is specially low: If the user completes the withdrawal later he will get the lowest price between the request price and the current price, as the code specifies. So the user will get a undesired amount of assets.

```
// As4626::_withdraw
// user gets the lowest price between the request price and the current price
uint256 price = (claimable >= _shares)
            ? AsMaths.min(last.sharePrice, request.sharePrice) // worst of if pre-
                existing request
            : last.sharePrice; // current price
```

If he tries to cancel the redeem instead, he will suffer a opportunity cost penalization if the share price has raised since the time of the request:

```
// As4626::cancelRedeemRequest
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);
}
```

2. Increase the request shares amount by any arbitrary amount of a already existing request in a moment where the shares price is lower than the request price, updating the request share price to a lower one.

```
// it will lower the price if the current price is lower than the request one
// example:
// last.sharePrice = 2
// request.sharePrice = 8
// request.shares = 10
// shares = 10000000000000000 can really request any amount

//              2*(10000000000000000 - 10)+(8*10)
// new price = ---------------------------------- = 2.000000006  => almost the
    current price
//                      1000000000000000000
 request.sharePrice =
                ((last.sharePrice * (shares - request.shares))
                 + (request.sharePrice * request.shares)) /
                shares;
```

Note that the same method can be used to inflate the request share price if the curret price is higher than the request price.

Recommendation: Do not allow to create requests on behalf of any ‘operator“ or require there is some kind of prior allowance.

Resolution: Resolved

6.2.3 Wrong repay calculation makes user repay the borrowed amount twice

Severity: *High risk*

Context: As4626.sol#L706

Description: In the `flashLoanSimple` function within `As4626.sol`, the calculation for the `toRepay` amount includes both the borrowed amount and the fees. However, during the check for the repaid amount, it calculates the difference between the current balance and the balance before the loan, expecting only the fees. This inconsistency results in the borrower being required to repay the borrowed amount plus fees, effectively doubling the repayment amount.

Proof of Concept:

```
// eg: amount is all the available balance
uint256 toRepay = amount + fee;
// eg: balance before is 10
uint256 balanceBefore = asset.balanceOf(address(this));
totalLent += amount;
// transfer the assets to borrower: 10
asset.safeTransferFrom(address(this), address(receiver), amount);
// receiver executes the operation and returns the borrowed amount + fees.
receiver.executeOperation(address(asset), amount, fee, msg.sender, params);
// requires current balance is balance before + amount + fee = 10 + 10 + fee
if ((asset.balanceOf(address(this)) - balanceBefore) < toRepay)
    revert FlashLoanDefault(msg.sender, amount);
```

Recommendation: The repayment check should consider only the fees, not the borrowed amount. Adjust the comparison as follows:

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

Resolution: Resolved

6.3 Medium risk

6.3.1 Potential inflation attack in As4626

Severity: *Medium risk*

Context: As4626.sol#L363

Description: The `seedLiquidity` function in the `As4626` contract is designed to be used as the initial liquidity deposit to prevent inflation attacks during the vault's initialization. The contract is paused in the constructor of `As4626Abstract` to secure the initialization. However, a potential vulnerability arises because the `mint` function, responsible for user deposits, does not include the `whenNotPaused` modifier. Consequently, users can deposit assets even when the contract is supposed to be paused during initialization. Additionally, the `maxTotalAssets` is set to `MAX_UINT256` by default, which does not impose a limit on the total assets, leaving room for potential inflation attacks.

Proof of Concept:

```
    // As4626.sol
    // no modifer
    function mint(
        uint256 _shares,
        address _receiver
    ) public returns (uint256 assets) {
        return _deposit(previewMint(_shares), _shares, _receiver);
    }

    // whenNotPaused modifier : cannot enter
    function deposit(
        uint256 _amount,
        address _receiver
    ) public whenNotPaused returns (uint256 shares) {
        return _deposit(_amount, previewDeposit(_amount), _receiver);
    }
```

Recommendation: To enhance security, add the `whenNotPaused` modifier to the mint function. Additionally, consider setting the `maxTotalAssets` to 0 during initialization for stricter control:

```
// As4626.sol
// Add whenNotPaused modifier to the mint function
function mint(
    uint256 _shares,
    address _receiver
) public whenNotPaused returns (uint256 assets) {
    return _deposit(previewMint(_shares), _shares, _receiver);
}

// Initialize maxTotalAssets to 0 for stricter control
function init(
    Erc20Metadata calldata _erc20Metadata,
    CoreAddresses calldata _coreAddresses,
    Fees calldata _fees
) public virtual onlyAdmin {
    // ... (other initialization code)
    maxTotalAssets = 0;
}
```

Resolution: Resolved

6.3.2 As4626 manager can lock users' funds

Severity: *Medium risk*

Description:

The `As4626` vault does not alow users to quit the vault if an emergency occurs. If the vault is paused users cannot liquidate their positions, menaing that their assets are not locked until the vault manager decides to do it.

Recommendation: Allow users to redeem/withdraw when the vault is paused.

Resolution: Akcnowledged

6.3.3 Using deprecated Chainlink function `latestAnswer`

Severity: *Medium risk*

Description:

The Chainlink utils functions calls the Chainlink oracles using the `latestAnswer` function. According to Chainlink's documentation, the `latestAnswer` function is deprecated. This function does not error if no answer has been reached but returns 0. Besides, the `latestAnswer` is reported with 18 decimals for crypto quotes but 8 decimals for FX quotes (See Chainlink FAQ for more details). A best practice is to get the decimals from the oracles instead of hard-coding them in the contract.

Proof of Concepr:

Here's the affected lines of the function `assetExchangeRate`:

```
 (uint256 quotePrice, uint256 basePrice) = (
            uint256(_priceFeeds[0].latestAnswer()),
            uint256(_priceFeeds[1].latestAnswer())
);
```

Recommendation:

Use the `latestRoundData` function to get the price instead. Add checks on the return data with proper revert messages if the price is stale or the round is uncomplete, for example:

```
// call
(uint80 roundID, int256 quotePrice, , uint256 timeStamp, uint80 answeredInRound) =
    _priceFeeds[0].latestRoundData();
//checks
require(answeredInRound >= roundID, "...");
require(timeStamp != 0, "...");
// call
(uint80 roundID2, int256 basePrice, , uint256 timeStamp2, uint80 answeredInRound2) =
     _priceFeeds[0].latestRoundData();
// checks
require(answeredInRound2 >= roundID, "...");
require(timeStamp2 != 0, "...");
```

Resolution: Resolved

6.3.4 Math rounding in `previewWithdraw` is not ERC4626-compliant

Severity: *Medium risk*

Context: As4626.sol#L446

Description: The `previewWithdraw` function in the `As4626` contract calculates the number of shares to burn for a withdrawal using the `convertToShares` function. The issue lies in the fact that `convertToShares` rounds down the result, which is not ERC4626-compliant. According to the ERC4626 standard, shares in `previewWithdraw` should be calculated by rounding up to prevent potential exploitation where users could withdraw more assets than intended due to a rounding error.

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

The issue is `convertToShares` rounds down the result:

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

Recommendation: Adjust the calculation in previewWithdraw to round up, ensuring ERC4626 compliance:

```
function previewWithdraw(uint256 _assets) public view returns (uint256) {
    return (_assets.mulDivUp(weiPerShare ** 2, sharePrice() * weiPerAsset)).addBp(
        exemptionList[msg.sender] ? 0 : fees.exit);
}
```

Resolution: Resolved

6.3.5 Math rounding in `previewMint` is not ERC4626-compliant

Severity: *Medium risk*

Context: As4626.sol#L425

Description: The `previewMint` function in the `As4626` contract calculates the number of shares to mint using the `convertToShares` function. However, the issue arises as `convertToShares` rounds down the result, which is not ERC4626-compliant. According to the ERC4626 standard, shares in `previewMint` should be calculated by rounding up to prevent potential exploitation where users could mint more shares than intended due to a rounding error.

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

The issue is `convertToShares` rounds down the result:

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

Recommendation: Adjust the calculation in previewMint to round up, ensuring ERC4626 compliance:

```
function previewMint(uint256 _assets) public view returns (uint256) {
    return (_assets.mulDivUp(weiPerShare ** 2, sharePrice() * weiPerAsset)).addBp(
        exemptionList[msg.sender] ? 0 : fees.exit);
}
```

Resolution: Resolved

6.3.6 `setSwapperAllowance` could revert with some tokens

Severity: *Medium risk*

Context: StrategyV5Agent.sol#L45C14-L45C33

Description: The `setSwapperAllowance` function in `StrategyV5Agent` has the potential to revert due to tokens with a race approval condition. Some tokens require specific conditions, such as the current allowance being zero, before approving a new allowance. This is commonly implemented to prevent users from frontrunning by reducing their allowance through a transfer. Tokens like USDT exhibit this behavior. In such cases, the contract may fail to approve the swapper, leading to a denial-of-service (DoS) scenario for certain functionalities.

Proof of Concept:

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

    for (uint256 i = 0; i < rewardLength; i++) {
        if (rewardTokens[i] == address(0)) break;
        // Potential race condition: If allowance != 0, reverts
        IERC20Metadata(rewardTokens[i]).approve(swapperAddress, _amount);
    }
    // ... (similar logic for other token types)
    asset.approve(swapperAddress, _amount);
}
```

Recommendation: To mitigate the race condition, reduce the allowance to zero before approving the new allowance.

```
for (uint256 i = 0; i < rewardLength; i++) {
    if (rewardTokens[i] == address(0)) break;
    // Bypass the race condition by setting allowance to 0 first
    IERC20Metadata(rewardTokens[i]).approve(swapperAddress, 0);
    // Set the actual allowance
    IERC20Metadata(rewardTokens[i]).approve(swapperAddress, _amount);
}
```

Resolution: Resolved

6.3.7 `redeem` and `withdraw` owner MUST always be the msg.sender

Severity: *Medium risk*

Context: As4626.sol#L245

Description: As per EIP-7540 the `redeem` and `withdraw` function's owner MUST always be the msg.sender.

Recommendation: Implement the following check:

```
function redeem(
        uint256 _shares,
        address _receiver,
        address _owner
    ) external returns (uint256 assets) {
        require(owner == msg.sender, "msg.sender is not the owner!")
        return _withdraw(previewRedeem(_shares), _shares, _receiver, _owner);
    }
```

Resolution: Resolved

6.4 Low risk

6.4.1 No callback support in requests

Severity: *Low risk*

Description: The ERC-7540 specifies that "All methods which initiate a request (including requestId==0) include a data parameter, which if nonzero length MUST send a callback to the receiver". Meaning it expects a callback to a caller `ERC7540RedeemReceiver` for the redeem/withdraw requests when the data is not empty.

Proof of Concept:

`https://eips.ethereum.org/EIPS/eip-7540#request-callbacks`

Recommendation: Implement the callback support as the EIP specifies in order to achieve fuch compatibility.

Resolution: Resolved

6.4.2 Incorrect use of `safetransferFrom`

Severity: *Low risk*

Context: As4626.sol#L711

Description: The contract intends to transfer its own funds by calling `transferFrom`

Resolution: Resolved

6.4.3 Cost of opportunity can ba inflated

Severity: *Low risk*

Context: As4626.sol#L577-L608

Description: Cost of opportunity can be inflated. When canceling a withdraw or redeem request if the share price has increased from the time you did the request the difference will be burnt from your shares. If someone sends funds to the vault(either by mistake or on purpose) it can increase the share price and therefore make the user burn more shares than he should.

Recommendation: Account the totalAssets internally.

Resolution: Aknowledged

6.4.4 Unsafe cast

Severity: *Low risk*

Context: StrategyV5Agent.sol#L111 | StrategyV5Agent.sol#L125

Description: The `setInput` and `setRewardTokens` functions could result in incorrect return if casting it to uint8 when the inputs array length is >= 256.

Recommendation: Use safeCast in order to prevent this issue.

Resolution: Resolved

6.4.5 Address(0) can lead to incorrect Chainlink update

Severity: *Low risk*

Context: StrategyV5Chainlink.sol#L53-L62

Description: If the array contains address(0) it will break the loop, so if there are addresses that are valid after that they will be ignored.

Recommendation: Consider skipping that iteration instead of breaking the loop in such case.

Resolution: Aknowledged

6.4.6 `maxDeposit` should return 0 when paused

Severity: *Low risk*

Context: As4626.sol#L462-L464

Description: The `maxDeposit` function in `As4626` MUST contain a `paused` check and return 0 when the contract is paused to return an accurate preview according to ERC4626.

Recommendation: To mitigate this issue and be comaptible with the standard, change the function as follows:

```
function maxDeposit(address) public view returns (uint256) {
        return
            paused()
                ? 0
                : maxTotalAssets.subMax0(totalAssets()
            );
    }
```

Resolution: Resolved

6.4.7 `maxWithdraw` should return 0 when paused

Severity: *Low risk*

Context: As4626.sol#L476

Description:

The `maxWithdraw` function in `As4626` MUST contain a `paused` check and return 0 when the contract is paused to return an accurate preview according to ERC4626.

Recommendation:

```
function maxWithdraw(address _owner) public view returns (uint256) {
    return paused() ? 0: convertToAssets(maxRedeem(_owner));
}
```

Resolution: Resolved

6.4.8 `maxMint` should return 0 when paused

Severity: *Low risk*

Context: As4626.sol#L706

Description:

The `maxMint` function in `As4626` MUST contain a `paused` check and return 0 when the contract is paused to return an accurate preview according to ERC4626.

Recommendation:

```
function maxMint(address) public view returns (uint256) {
    return paused()? 0 : convertToShares(maxDeposit(address(0)));
}
```

Resolution: Resolved

6.4.9 Missing `claimableRedeemRequest` function

Severity: *Low risk*

Description:

`As4626`misses the `claimableRedeemRequest` view function from the ERC-7540 standard

Recommendation: Add the function to the code so it complies with the standard

Resolution: Resolved

6.4.10 `requestDeposit` function not implemented

Severity: *Informational*

Context: As4626.sol#L501

Description: For some reason the `requestDeposit` function has been left commented, but is present in the IAs4626 interface. This could lead to a lot of complications dues to EIP-7540 incompatibility as this function is the entry point of the flow of this standard. Also when other contracts are interacting with this one, they would expect a certain functionality but will not meet it and this can lead to confusion and unexpected behaviour.

Recommendation: Implement `requestDeposit` function according to EIP-7540

Resolution: Not resolved

6.5 Informational

6.5.1 Entry fees might desincentivize users to deposit

Severity: *Low risk*

Description: The entry fees desincentivize investment, since users will incurr losses by just depositing and will need to wait for profit just to recover the initial amount.

Recommendation: Consider removing the entry fees.

Resolution: Aknowledged

6.5.2 Consider using specific imports

Severity: *Informational*

Description: Import only the contracts, structs etc that are strictly necessary from other files, instead of importing the whole file , this will improve the readability and size of the contracts.

Resolution: Aknowledged

6.5.3 Non-reentrant modifier should be placed before every other modifier

Severity: *Informational*

Context: StrategyV5.sol#L128

Description: As per solodit checklist, nonReentrant modifier should ALWAYS be placed first.

Resolution: Aknowledged

6.5.4 Confusing naming in `last.accountedAssets` and `last.accountedSupply`

Severity: *Informational*

Description: : This values store the latest `totalAssets()` instead of `totalAccountedAssets()` and `totalSupply()` instead of `totalAccountedSupply()`. This naming is very confusing.

Recommendation: name them "last.totalSupply" and "last.totalAssets"

Resolution: Aknowledged

6.5.5 Consider adding a custom error message in `exchangeRate`

Severity: *Informational*

Context: As4626.sol#L501

Description: : The `exchangeRate` function in `AsMath` has a `require` statement in it with no custom error message, leading to a more difficult debugging.

Recommendation: Add a custom string error message

Resolution: Aknowledged

6.5.6 Use 0xEeee to represent native assets

Severity: *Informational*

Context: AsRescuable.sol#L83

Description: : In the `_rescue` function in `AsRescuable` `address(1)` is used to represent native assets, while using the `0xEeeeeEeeeEeEeeEeEeeeEEEeeeeEeeeeeeeEEeE` address is a more standard value for this matter, according to EIP-7528

Recommendation: Use `0xEeeeeEeeeEeEeeEeEeEeeEEEeeeeEeeeeeeeEEeE` instead of `address(1)`

Resolution: Aknowledged

6.5.7 Uninitialized `asset` and `weiPerAsset` values in the 'As4626" abstract contract

Severity: *Informational*

Description: : `assets` is meant to be initialized later by a inheriting contract, but it would make more sense initializing in the `As4626` contract itself, the same way it is initialized in ERC4626. For the same reason, initializing `weiPerAsset` as well would make sense, since its used to calculate the share price and perform the shares to assets and opposite conversions.

Recommendation: Initialize it in the `init` function of As4626 :

```
    function init(
        Erc20Metadata calldata _erc20Metadata,
        CoreAddresses calldata _coreAddresses,
        Fees calldata _fees
    ) public virtual onlyAdmin {
        // check that the fees are not too high
        setFees(_fees);
        feeCollector = _coreAddresses.feeCollector;
        req.redemptionLocktime = 6 hours;
        last.accountedSharePrice = weiPerShare;
        last.accountedProfit = weiPerShare;
        last.feeCollection = uint64(block.timestamp);
        last.liquidate = uint64(block.timestamp);
        last.harvest = uint64(block.timestamp);
        last.invest = uint64(block.timestamp);
        ERC20._init(_erc20Metadata.name, _erc20Metadata.symbol, _erc20Metadata.
            decimals);

        // initialize asset
        asset = IERC20Metadata(_coreAddresses.wgas);

        // initialze weiPerAsset
        weiPerAsset = 10 ** asset.decimals();
    }
```

Resolution: Aknowledged

6.5.8 Unnecessay check in `_deposit`

Severity: *Informational*

Description: : In the `_deposit` function of `As4626` it checks that the shares is not more than the assets converted to shares, according to the current `sharePrice`. But this check should not be necessary if `previewMint` and `previewWithdraw` are correctly implemented,as the `deposit` and `mint`functions fetch it at that right moment from `previewDeposit` and `previewMint`.

```
function _deposit(
        uint256 _amount,
        uint256 _shares,
        address _receiver
    ) internal nonReentrant returns (uint256) {

        if (_receiver == address(this) || _amount == 0) revert Unauthorized();

        // do not allow minting at a price higher than the current share price
        last.sharePrice = sharePrice();

        if (_amount > maxDeposit(address(0)) || _shares > _amount.mulDiv(weiPerShare
             ** 2, last.sharePrice * weiPerAsset))
            revert AmountTooHigh(_amount);
```

Recommendation: Remove the unnecessary check.

Resolution: Aknowledged

6.5.9 Consider using a fixed solidity version

Severity: *Informational*

Description: : The code is currently using pragma `^0.8.0`, giving the possibility to compile it with a wide range of solidity versions, including versions that may have some compiler bugs or similar.

Recommendation: Consider using a fixed stable soliidity version.

Resolution: Aknowledged

6.5.10 Missing `safeMint` function

Severity: *Informational*

Description: : The `As4626` contract has `safeDeposit`, `safeRedeem` and `safeWithdraw` functions but for some reason `safeMint` is missing.

Recommendation: Add the `safeMint` function so it is consistant.

Resolution: Aknowledged

6.5.11 Users can do redeem requests when the As4626 contract is paused

Severity: *Informational*

Description: : The users are able to create redeem requests when the contract is paused, meaning that if that doesnt change users won't be able to actually redeem.

Recommendation: Consider not allowing requests when the vault is paused

Resolution: Aknowledged