code4rena

Strateg

Smart Contract Security Assessment

Version 2.0

Audit dates: Jun 03 — Jun 17, 2024

Audited by: MiloTruck
thereksfour
IAm0x52

Contents

1. Introduction

1.1 About Zenith

Zenith is an offering by Code4rena that provides consultative audits from the very best security researchers in the space. We focus on crafting a tailored security team specifically for the needs of your codebase.

Learn more about us at https://code4rena.com/zenith.

1.2 Disclaimer

This report reflects an analysis conducted within a defined scope and time frame, based on provided materials and documentation. It does not encompass all possible vulnerabilities and should not be considered exhaustive.

The review and accompanying report are presented on an "as-is" and "as-available" basis, without any express or implied warranties.

Furthermore, this report neither endorses any specific project or team nor assures the complete security of the project.

1.3 Risk Classification

SEVERITY LEVEL	IMPACT: HIGH	IMPACT: MEDIUM	IMPACT: LOW
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

2. Executive Summary

2.1 About Strateg

The infrastructure for creating and sharing complex strategies, leveraging the most innovative DeFi protocols.

2.2 Scope

Repository	strateg-protocol/strateg-protocol-core
Commit Hash	0xc5bd035c4529fe94776c3e698e7beb6877c5a6befbd78f030241a30ccc4eb5c0
Mitigation Hash	df2fa55e6cf9e9c6d9305064b36a0f1065106425

2.3 Audit Timeline

DATE	EVENT
Jun 03, 2024	Audit start
Jun 17, 2024	Audit end
Oct 28, 2024	Report published

2.4 Issues Found

SEVERITY	COUNT
Critical Risk	0
High Risk	4
Medium Risk	12
Low Risk	10
Informational	3
Total Issues	29

3. Findings Summary

ID	DESCRIPTION	STATUS
H-1	All rewards in `StrategERC3525` can be stolen with a single token	Resolved

H-2	Precision loss in `withdrawalRebalance()` will make users lose funds	Resolved
H-3	Donation attacks can inflate the ratio of assets/shares	Resolved
H-4	Duplicate strategies in Vault cause assets to be counted repeatedly	Resolved
M-1	Missing `_bufferDerivation <= _bufferSize` in `StrategVaultFactory.deployNewVault()` could DOS rebalancing	Resolved
M-2	`StrategVault.withdrawalRebalance()` can be DOSed by `_harvestFees()` when remaining assets in the buffer is small	Resolved
M-3	`StrategVault._applyTvlLimits()` incorrectly assumes `msg.sender` is the depositor	Resolved
M-4	`_decreaseValueDeposited` is applied to incorrect user in multiple locations which can lead to deposit DOS	Resolved
M-5	Holder, whitelist and TVL requirements can be bypassed by minting or sending shares to alternate address	Resolved
M-6	User can bypass timelock by minting shares to alternate receiver	Resolved
M-7	Users can frontrun `_harvestFees()` to withdraw assets to avoid losses.	Resolved
M-8	`_fetchNativeTVL(false)` causes stale nativeTvl to be used.	Resolved
M-9	Incorrect check in `_applyTvlLimits()`	Resolved
M-10	`signedHash` should contain `_vault` in `vaultWithdrawalRebalance()`	Resolved
M-11	`putInBuffer()` doesn't work with rebase tokens.	Resolved
M-12	A malicious user can frontrun permit transaction to make it revert due to invalid signature	Resolved

L-1	Missing array length checks in `StrategVaultFactory.editVaultParams()`	Resolved
L-2	Missing `_vault` address validation in `StrategVaultFactory `functions	Resolved
L-3	`StrategVault.redeem()` can be forced to revert if the shares redeemed is close to `bufferShares`	Acknowledged
L-4	Exit hook is not called when performing a withdrawal in `StrategVault.withdrawalRebalance()`	Resolved
L-5	`minUserDeposit` limit in `StrategVault` can be bypassed	Resolved
L-6	`StrategVault#_rebalance` fails to clear highly sensitive dynamic parameters when exiting to buffer	Resolved
L-7	StrategVault#withdraw and redeem applies timelock to sender which can lead to incorrectly reverting transactions	Resolved
L-8	_rebalance does not consider assets in the vault	Resolved
L-9	The pending settings cannot be canceled.	Resolved
L-10	`removeTokenPercent()` and `removeAllTokenPercent()` do not work as described	Resolved
I-1	Specifying `gasleft()` as the gas limit in `StrategVaultFactory.vaultDeposit()` is redundant	Resolved
I-2	Missing length checks on `_dynParamsIndex` and `_dynParams` in `StrategVault._storeDynamicParams()`	Resolved
I-3	Calls to `maxMint` and `maxDeposit` as validation during deposits and mints are unnecessary	Resolved

4. Findings

4.1 High Risk

A total of 4 high risk findings were identified.

[H-1] All rewards in `StrategERC3525` can be stolen with a single token

Severity: High	Status: Resolved

Context:

- StrategERC3525.sol#L258-L260
- StrategERC3525.sol#L192-L194

Description:

In `_claimRewards()`, `info.lastClaimTotalRewards` is only updated for a user when `feeClaimable`, which is the amount of rewards currently accrued to the user, is more than `0`:

```
if (feeClaimable > 0) {
    info.timeClaimed = block.timestamp;
    info.lastClaimTotalRewards = totalRewards;
```

However, not updating `info.lastClaimTotalRewards` when `feeClaimable = 0` allows for a `tokenId`'s balance to change while `lastClaimTotalRewards` remains outdated.

`_claimRewards()` is called in `_beforeValueTransfer()` whenever value is transferred to an existing `tokenid`:

```
if (_exists(_toTokenId)) {
    _claimRewards(_toTokenId);
} else {
```

As such, if value is transferred to an existing `tokenId` that has a zero balance, that `tokenId` instantly gains rewards. This effectively creates rewards out of thin air, for example:

- Assume a user has two token IDs:
  - `tokenId = 1` has a balance of 500 (half of `RATIO`).
  - `tokenId = 2` has a balance of 0.
  - `info.lastClaimTotalRewards` is currently `0` for both tokens.

- Vault calls `addRewards()` to add 100 tokens of rewards.
- User claims 50 tokens from `tokenId = 1`.
- User calls `transferFrom()` with `value_ = 500` to transfer value from `tokenId = 1` to `tokenId = 2`.
- In the `_beforeValueTransfer()` hook:
  - `tokenId = 2` exists, so `_claimedRewards()` is called.
  - `feeClaimable = 0` as `tokenId = 2` has no balance.
  - `info.lastClaimTotalRewards` is not updated.
- Now, when `_claimRewards()` is called again for `tokenId = 2` (eg. through `redeem()`):
  - `concernedRewards = 100 - 0 = 100`
  - `feeClaimable = 100 * 500 / 1000 = 50`
  - The user gets to claim 50 tokens again, even though they were already claimed in `tokenId = 1`.

This can be repeatedly performed to steal all rewards in the contract.

Note that a `tokenId` with zero balance can be created by calling `transferFrom()` with `value_ = 0` to a new address.

Recommendation:

Consider updating `info.lastClaimTotalRewards` regardless of `feeClaimable`:

```
+ info.timeClaimed = block.timestamp;
+ info.lastClaimTotalRewards = totalRewards;
  if (feeClaimable > 0) {
-     info.timeClaimed = block.timestamp;
-     info.lastClaimTotalRewards = totalRewards;
```

Strateg: Fixed with the following commit

Zenith: Verified.

[H-2] Precision loss in `withdrawalRebalance()` will make users lose funds

Severity: High	Status: Resolved

Context:

- StrategVault.sol#L521-L550

Description:

Users may suffer losses due to precision loss in `withdrawalRebalance()`. Consider totalSupply = 100000e18, amount = 19e18. withdrawPercent = 19e18 * 10000/100000e18 = 1.9, round down to 1. In the end, users will only withdraw 0.01% of totalAssets, but 0.019% of totalShares will be burned.

```
        uint256 withdrawPercent = (_amount * 10000) / totalSupply();
        if(withdrawPercent == 10000) {
            _harvestFees(false);
        }

        _exitStrategy(withdrawPercent);
        uint256 totalBufferWithdraw = withdrawPercent *
_asset.allowance(_buffer, address(this)) / 10000;
        _asset.safeTransferFrom(
            _buffer,
            address(this),
            totalBufferWithdraw
        );

        uint256 tAssets = _fetchNativeTVL(false);
        totalWithdraw = tAssets *  withdrawPercent / 10000;

        uint256 currentBalance = _asset.balanceOf(address(this));
        if(currentBalance < totalWithdraw) {
            uint256 diffPercent = (totalWithdraw - currentBalance) *
10000 / totalWithdraw;
            if(diffPercent < 100) { //less than 1% spread
                totalWithdraw = currentBalance;
            } else {
                revert WithdrawalRebalanceIssue();
            }
        }

        /**
         * @dev Execute user withdrawal with ERC4626 low level function
```

```
         */
        _burn(msg.sender, _amount);
```

Less than 0.01 % is lost in `withdrawalRebalance()`. And even if the user is aware of this, it may be lost due to earlier transactions by other users. Consider Alice deposits 0.01%, Bob withdraws 0.03%, and Alice's transaction is executed first, so Bob's 0.03% becomes 0.03%/(1+0.01%) = 0.0299%, and Bob withdraws only 0.02% assets.

Recommendation:

Instead of burning `_amount`, it is recommended to burn `totalSupply * withdrawPercent / 10000`.

```
-       _burn(msg.sender, _amount);
+       _burn(msg.sender, mulDiv(totalSupply(), withdrawPercent, 10000,
true)); // round up
```

Strateg: The recommendation has been implemented: StrategVault.sol#L648

Zenith: Verified.

[H-3] Donation attacks can inflate the ratio of assets/shares

Severity: High	Status: Resolved

Context:

- StrategVault.sol#L824-L842

Description: The first depositor can make a donation attack, which will inflate the ratio of assets/shares. In `_harvestFees()`, `taxableValue` is based on the portion of funds that have appreciated. `lastHarvestIndex` is initially 10000, which means that it is assumed that assets/shares = 1:1 at the beginning. If the first depositor inflate the ratio of assets/shares by donation attack, for example, inflate the ratio to 10000:1, i.e. `currentVaultIndex = 100000000`, then `_harvestFees()` will charge fees based on total assets. A malicious deployer have incentive to do so in order to overcharge fees.

```
    function _harvestFees(bool _applyHarvestFee) private {
        _fetchNativeTVL(false);
        DataTypes.VaultConfigurationMap memory config =
_getConfiguration();
        uint256 tSupply = totalSupply();
        uint256 _lastFeeHarvestIndex = lastHarvestIndex;
        uint256 tAssets = totalAssets();
        uint256 currentVaultIndex = (tAssets * 10000) / tSupply;


        uint256 _protocolFee = config.getProtocolFee();
        uint256 _creatorFee = config.getCreatorFee();
        uint256 _harvestFee = config.getHarvestFee();

        if (_lastFeeHarvestIndex == currentVaultIndex ||
currentVaultIndex < _lastFeeHarvestIndex) {
            lastHarvestIndex = currentVaultIndex;
            return;
        }

        uint256 taxableValue = (tSupply * currentVaultIndex / 10000) -
(tSupply * _lastFeeHarvestIndex / 10000);
```

Also, donation attack can be used to bypass the `maxUserDeposit` limit. Consider `maxUserDeposit` is 1e18 assets, the user deposits 1e17 assets, and gets 1e17 shares. The user sends 1e20 assets to the Vault, and totalAssets() is 1e20+1e17. The user's 1e17 shares will be worth 1e17 * (1e20 + 1e17 + 1) / (1e17 + 1) asseets, which is basically no loss.

Recommendation: Consider getting rid of the tracking of `_asset.balanceOf(address(this))` in `_getNativeTVL()`, since the protocol by design doesn't keep the asset in the vault, but sends it to the buffer.

Strateg: We reworked the fees system and the tracking of the tvl. the lastVaultIndex has been replaced by vaultIndexHighWaterMark that represent the highest vaultIndex on which fees has been taken. The vaultIndex is now calculated on each deposit and fees are taken instantly before processing shares to mint. StrategVault.sol#L928

Additionally, we added a protection on deposit against rounding loss to avoid loss on donation frontrun attack, it wwas the only way we found to pass the donation attack fuzzing test StrategVault.sol#L254

Zenith: Verified.

[H-4] Duplicate strategies in Vault cause assets to be counted repeatedly

Severity: High	Status: Resolved

Context:

- StrategVault.sol#L362-L411

Description:

Protocol allows vault to set duplicate strategies, but this can lead to repeated counting of assets in the strategy, malicious deployers can exploit it to inflate total assets. For example, the malicious deployer sets up 5 strategies with the same address and deposits 1000 tokens A to get 1000 shares, and then enters the strategies. The first strategy converts Token A to Token B. The next four strategies do not perform any action because the balance of Token A is 0. Then in `_getNativeTVL()`, `oracleExit()` of all five strategies will use Token B's balance to calculate, which means Token B's balance will be counted 5 times repeatedly.

```
    function _getNativeTVL() internal view returns (uint256) {
        address _asset = asset();

        DataTypes.OracleState memory oracleState;
        oracleState.vault = address(this);

        uint256 _strategyBlocksLength = strategyBlocksLength;
        if (_strategyBlocksLength == 0 || !isLive) {
            return IERC20(_asset).balanceOf(address(this)) +
IERC20(_asset).allowance(buffer, address(this));
        } else if (_strategyBlocksLength == 1) {
            oracleState =
                IStrategStrategyBlock(strategyBlocks[0]).oracleExit(
                    oracleState,
                    LibBlock.getStrategyStorageByIndex(0),
                    10000
                );
        } else {
            uint256 revertedIndex = _strategyBlocksLength - 1;
            for (uint256 i = 0; i < _strategyBlocksLength; i++) {
                uint256 index = revertedIndex - i;
                oracleState =
IStrategStrategyBlock(strategyBlocks[index]).oracleExit(
                    oracleState,
LibBlock.getStrategyStorageByIndex(index), 10000
                );
```

```
            }
        }
```

At this point totalAsset() is 5000, and when other users deposit 1000 tokens of A, they will only get 200 share. Exiting the strategy, the attacker will have (1000+1000)*1000/(1000+200) = 1667 token A.

Recommendation:

Since the protocol supports Vault setting duplicate strategies, it is recommended to remove duplicate token counts in `oracleExit()` of the strategies.

Strateg: The oracleExit function keep a list of all tokens address manipulated by the vault strategy. On this block, we'll always use the tokenExists function that has been added in 0.1.17 version of the strateg-protocol-libraries: LibOracleState.sol#L35 It will avoid to encounter this duplication issue

Zenith: Verified.

4.2 Medium Risk

A total of 12 medium risk findings were identified.

[M-1] Missing `_bufferDerivation <= _bufferSize` in `StrategVaultFactory.deployNewVault()` could DOS rebalancing

Severity: Medium	Status: Resolved

Context:

- StrategVaultFactory.sol#L710-L716
- StrategVault.sol#L716-L719
- StrategVaultFactory.sol#L190-L191

Description:

An invariant of all vaults is that `_bufferDerivation` has to be not greater than `_bufferSize`, which is enforced in `_setVaultBufferParams()` through the following check:

```
if (
    _bufferSize > 9000 ||
    _bufferSize < 300 ||
    _bufferDerivation < 200 ||
    _bufferDerivation > 2000 ||
    _bufferSize < _bufferDerivation
) revert BadBufferParams();
```

This ensures that if the buffer is undersized, calculating `bufferSize - derivation` in `StrategVault.rebalance()` does not revert:

```
/**
 * Buffer undersized
 */
if (currentBufferSize < bufferSize - derivation) {
```

However, in `StrategVaultFactory.deployNewVault()`, there is no check that ensures `_bufferSize` is greater than or equal to the default buffer derivation of `500`:

```
c.setBufferSize(_bufferSize);
c.setBufferDerivation(500);
```

As such, if `_bufferSize` is smaller than `500` on vault deployment, `rebalance()` will always revert when called while the vault's buffer is undersized, making rebalancing impossible.

Recommendation:

In `StrategVaultFactory.deployNewVault()`, check that `_bufferSize` is larger than or equal to `500`.

Strateg: Here we have removed the buffer configuration on the deploy function to force it to bufferSize = 1000 and bufferDerivation = 500 to avoid duplicated code and ensure a valid buffer configuration on vault deployment

Zenith: Verified.

[M-2] `StrategVault.withdrawalRebalance()` can be DOSed by `_harvestFees()` when remaining assets in the buffer is small

Severity: Medium	Status: Resolved

Context:

- StrategVault.sol#L847-L858
- StrategVault.sol#L522-L524

Description:

In `_harvestFees()`, fees are taken from the buffer and transferred to their respective recipients:

```
//creatorFee
uint256 creatorFeeAmount = (taxableValue * _creatorFee) / 10000;
_asset.safeTransferFrom(buffer, address(this), creatorFeeAmount);
_asset.safeIncreaseAllowance(erc3525, creatorFeeAmount);
IStrategERC3525(erc3525).addRewards(creatorFeeAmount);

//harvestFee
if(_applyHarvestFee)
    _asset.safeTransferFrom(buffer, msg.sender, (taxableValue *
_harvestFee) / 10000);

//protocolFee
_asset.safeTransferFrom(buffer, addressProvider.feeCollector(),
(taxableValue * _protocolFee) / 10000);
```

However, if the buffer has too little assets remaining, it becomes possible for `_harvestFees()` to revert when attempting to collect fees.

This becomes an issue in `withdrawalRebalance()`, where `_harvestFees()` is called when attempting to withdraw all shares:

```
if(withdrawPercent == 10000) {
    _harvestFees(false);
}
```

If the remaining amount of assets in the buffer is smaller than the total amount of fees, `_harvestFees()` will always revert, causing the function to be DOSed until `rebalance()` is

called by operator to refill the buffer.

Recommendation:

Instead of taking fees from the buffer whenever `_harvestFees()` is called, consider accumulating the amount of fees in a state variable. The fees should only be taken from the buffer and paid out when `harvestFees()` is called by the operator.

This ensures that no functions will be DOSed when the buffer has a small amount of assets remaining. Additionally, to resolve the situation the operator can simply call `rebalance()` to refill the buffer before calling `harvestFees()`.

Strateg: The recommendation has been implemented. Now on each function modifying the vault total supply, the _updateIndex function is called.

- StrategVault.sol#L928

This function calculates fees on the flight and store it on the state. Now the harvest function executes the harvest block execution, executes the _updateIndex, and distributes the fees

Zenith: Verified - with the new change, fees are accumulated in taxValueCumulated and only distributed when harvest() is called.

[M-3] `StrategVault._applyTvlLimits()` incorrectly assumes `msg.sender` is the depositor

Severity: Medium	Status: Resolved

Context:

- StrategVault.sol#L918
- StrategVault.sol#L223-L226

Description:

In `_applyTvlLimits()`, the total deposit amount the depositor has is fetched from the `userDeposit` mapping using `msg.sender`:

```
uint256 userDeposited = userDeposit[msg.sender];
```

However, this is incorrect as `deposit()`/`mint()` is called through the factory, `msg.sender` would be the factory address instead of the user. Using `deposit()` as an example, the user is determined through `_msgSender()` instead of `msg.sender`:

```
address sender = _msgSender();
address assetProvider =
    msg.sender != factory ?
    msg.sender : addressProvider.userInteractions();
```

As such, the minimum and maximum user deposit checks will be performed with the wrong `userDeposited` amount, allowing these checks to be bypassed.

Recommendation:

This has been fixed by the team by removing the `userDeposit` mapping amongst other changes.

Strateg: Resolved.

Zenith: Verified.

[M-4] `_decreaseValueDeposited` is applied to incorrect user in multiple locations which can lead to deposit DOS

Severity: Medium	Status: Resolved

Context

StrategVault.sol#L276-L293

StrategVault.sol#L302-L320

StrategVault.sol#L507-L563

Description

StrategVault.sol#L315-L320

```
        uint256 assets = previewRedeem(_shares);
        _withdraw(sender, _receiver, __owner, assets, _shares);
        _decreaseValueDeposited(sender, _shares);

        return assets;
    }
```

When redeeming or withdrawing, `userDeposit` is decreased for the sender rather than the `owner` providing the shares. `userDeposit` is used to determine user specific deposit limits. This will inflate `userDeposit` for `owner` (since their withdraw is not credited back to them) and incorrectly prevents them from depositing more due to deposit limits.

StrategVault.sol#L550-L553

```
        _burn(msg.sender, _amount);
        _asset.safeTransfer(_user, totalWithdraw);
        emit Withdraw(msg.sender, msg.sender, msg.sender, totalWithdraw,
    _amount);
        _decreaseValueDeposited(_user, _amount);
```

Additionally `withdrawalRebalance` applies `_decreaseValueDeposited` to `_user` which in this case is always the operator proxy. This leads to the same issue as outlined above.

Recommendation

As discussed, user deposit limits will be removed

Strateg: Resolved

Zenith: Verified.

[M-5] Holder, whitelist and TVL requirements can be bypassed by minting or sending shares to alternate address

Severity: Medium	Status: Resolved

Context StrategVault.sol#L1000-L1004

StrategVault.sol#L1009-L1015

StrategVault.sol#L905-L941

Description

StrategVault.sol#L946-L950

```
function _applyWhitelisted() internal view {
    if (

!IStrategVaultFactory(factory).addressIsWhitelistedOnVault(address(this),
_msgSender())
    ) revert NotWhitelisted();
}
```

`_applyWhitelist` is only checks that `sender` is on the whitelist allowing them to set the receiver to a non-whitelisted address, bypassing whitelisting. The same applies to `_applyHolder` and `_applyTvlLimits`.

StrategVault.sol#L864-L871

```
function _update(address from, address to, uint256 value) internal
override {
    _applyTimelock(from);
    super._update(from, to, value);
    emit StrategVaultUpdate(
        StrategVaultUpdateType.Transfer,
        abi.encode(from, to, value)
    );
}
```

Additionally, middleware is not applied at all during update allowing easy bypass of these limits by simply sending the shares to another address.

Recommendation

_applyMiddleware should be called on _update and `_applyWhitelist` `_applyHolder` and `_applyTvlLimits` should apply to the user receiving the shares, rather than the sender.

Strateg: This checks are done on _update now

Zenith: Verified

[M-6] User can bypass timelock by minting shares to alternate receiver

Severity: Medium	Status: Resolved

Context: StrategVault.sol#L219-L241

StrategVault.sol#L249-L267

Description

Users can bypass timelock on shares by minting to a separate `receiver`.

StrategVault.sol#L261-L264

```
        _applyMiddleware(assets, vaultTVL, config);
        _resetTimelock(sender, config);
        _deposit(sender, receiver, assets, shares);
        _incrementValueDeposited(sender, shares);
```

When depositing, _resetTimelock is always called with sender address rather than receiver address. This fails to apply the timelock to `receiver`, allowing them to immediately transfer or redeem the tokens.

Recommendation

Timelock should be set on `receiver` rather than `sender`

Strateg: These checks are done on _update now and based on the receiver: StrategVault.sol#L957

Additionally, following the recommendation on PR comments, a weighted calculation is done on transfer and mint to avoid withdrawing DOS: StrategVault.sol#L1111

Zenith: Middleware is now applied on _update, meaning that it is always applied to the receiver of the shares. Additionally, while DOS attacks are still technically possible it would require 1/timelock * funds locked compounded, per withdrawal attempt blocked. This makes DOS attempts impractical and costly to the attacker.

[M-7] Users can frontrun `_harvestFees()` to withdraw assets to avoid losses.

Severity: Medium	Status: Resolved

Context:

- StrategVault.sol#L824-L859

Description:

`_harvestFees()` will reduce `totalAsset()`, and users can frontrun _harvestFees() to withdraw assets to avoid losses.

```
    function _harvestFees(bool _applyHarvestFee) private { // 会减小
totalAsset ,
        _fetchNativeTVL(false);
        DataTypes.VaultConfigurationMap memory config =
_getConfiguration();
        uint256 tSupply = totalSupply();
        uint256 _lastFeeHarvestIndex = lastHarvestIndex;
        uint256 tAssets = totalAssets();
        uint256 currentVaultIndex = (tAssets * 10000) / tSupply;


        uint256 _protocolFee = config.getProtocolFee();
        uint256 _creatorFee = config.getCreatorFee();
        uint256 _harvestFee = config.getHarvestFee();

        if (_lastFeeHarvestIndex == currentVaultIndex ||
currentVaultIndex < _lastFeeHarvestIndex) {
            lastHarvestIndex = currentVaultIndex;
            return;
        }

        uint256 taxableValue = (tSupply * currentVaultIndex / 10000) -
(tSupply * _lastFeeHarvestIndex / 10000);
        uint256 totalFeeDistributed = (taxableValue * _creatorFee /
10000) + (_applyHarvestFee ? taxableValue * _harvestFee / 10000 : 0) +
(taxableValue * _protocolFee / 10000);
        lastHarvestIndex = (tAssets - totalFeeDistributed) * 10000 /
tSupply;
        IERC20 _asset = IERC20(asset());

        //creatorFee
        uint256 creatorFeeAmount = (taxableValue * _creatorFee) / 10000;
```

```
        _asset.safeTransferFrom(buffer, address(this), creatorFeeAmount);
        _asset.safeIncreaseAllowance(erc3525, creatorFeeAmount);
        IStrategERC3525(erc3525).addRewards(creatorFeeAmount);

        //harvestFee
        if(_applyHarvestFee)
            _asset.safeTransferFrom(buffer, msg.sender, (taxableValue *
_harvestFee) / 10000);

        //protocolFee
        _asset.safeTransferFrom(buffer, addressProvider.feeCollector(),
(taxableValue * _protocolFee) / 10000);
    }
```

Recommendation:

It is recommended that `_harvestFees()` should be called before each deposit and withdrawal. In order to save gas, we can save the fee to be collected in a state variable and subtract it from totalAsset, then the operator can collect it.

Strateg: The recommendation has been implemented with this commit. Now on each function modifiying the vault total supply, the _updateIndex function is called.

Zenith: Verified.

[M-8] `_fetchNativeTVL(false)` causes stale nativeTvl to be used.

Severity: Medium	Status: Resolved

Context:

- StrategVault.sol#L778-L785

Description: `_fetchNativeTVL(false)` will only update nativeTvl when `block.timestamp > lastNativeTVLUpdate`, if `mint/deposit` is executed multiple times in one block, the vaultTVL gotten by `_fetchNativeTVL(false)` will be stale.

```
    function _fetchNativeTVL(bool _force) internal returns (uint256) {
        if (lastNativeTVLUpdate == block.timestamp && !_force) return
nativeTVL;

        uint256 tvl = _getNativeTVL();
        nativeTVL = tvl;
        lastNativeTVLUpdate = block.timestamp;
        return tvl;
    }
```

Consider that the current VaultTVL is 10000. In the same block, Alice calls `deposit()` to deposit 1000 and gets the VaultTVL of 10000, `lastNativeTVLUpdate = block.timestamp`. Alice calls `deposit()` the second time and deposits 1000, and since `_force` is false and `lastNativeTVLUpdate == block.timestamp`, `_fetchNativeTVL(false)` returns the cached 10000 instead of the latest 11000, which allows the user to bypass the `maxVaultDepositcheck`.

Also in `withdrawalRebalance()`, if `stratBlocksLength == 0`, `_fetchNativeTVL(true)` in `_exitStrategy()` will not be called, then `_fetchNativeTVL(false)` after that will return the stale nativeTvl.

```
        _exitStrategy(withdrawPercent);
        uint256 totalBufferWithdraw = withdrawPercent *
_asset.allowance(_buffer, address(this)) / 10000;
        _asset.safeTransferFrom(
            _buffer,
            address(this),
            totalBufferWithdraw
        );
```

```
            uint256 tAssets = _fetchNativeTVL(false);
            totalWithdraw = tAssets *  withdrawPercent / 10000;
```

If the attacker frontruns `withdrawalRebalance()` to deposit a large amount of assets, the withdrawPercent will decrease due to the increase in `totalSupply()`, but since the stale VaultTVL returned by `_fetchNativeTVL(false)` is used here, this will reduce the asset withdrawn by the user.

Recommendation: It is recommended to update nativeTvl in `deposit/mint/redeem/withdraw/withdrawalRebalance/harvest` to make it latest.

Strateg: The recommendation has been implemented StrategVault.sol#L928. Now on each function modifiying the vault total supply, the _updateIndex function is called. The _updateIndex function update the nativeTVL variable

Zenith: Verified.

[M-9] Incorrect check in `_applyTvlLimits()`

Severity: Medium	Status: Resolved

Context:

- StrategVault.sol#L905-L930

Description: `_applyTvlLimits()` performs deposit limit checking.

```
    function _applyTvlLimits(uint256 _amount, uint256 _vaultTvl,
DataTypes.VaultConfigurationMap memory config) internal view {
        uint256 mode = config.getLimitMode();

        //No limit set
        if (mode == 0) return;

        (
            uint256 minUserDeposit,
            uint256 maxUserDeposit,
            ,
            uint256 maxVaultDeposit
        ) =
IStrategVaultFactory(factory).getVaultDepositLimits(address(this));

        uint256 userDeposited = userDeposit[msg.sender];

        //Minimum limit set
        if (mode % 2 == 1) {
            uint256 minDeposit = minUserDeposit;
            if (userDeposited == 0) {
                if (_amount < minDeposit) revert MinDepositNotReached();
            } else {
                if (userDeposited + _amount < minDeposit) {
                    revert MinDepositNotReached();
                }
            }
        }

        //Vault max limit set
        if (mode >= 4 && _vaultTvl + _amount > maxVaultDeposit) {
            revert MaxVaultDepositReached();
        }

        //User max limit set
```

```
            if (mode % 4 > 1 && userDeposited + _amount > maxUserDeposit) {
                revert MaxUserDepositReached();
            }
        }
```

The problem here is that `userDeposit` represents the amount of shares the user has, while `_applyTvlLimits()` checks for the amount of assets deposited. Since assets and shares are not 1:1, this check is incorrect.

Recommendation:

It is recommended to make the checking consistent so that `_applyTvlLimits()` checks for only assets or shares.

Strateg: All checks are based on shares now - StrategVault.sol#L957

Zenith: Verified.

[M-10] `signedHash` should contain `_vault` in `vaultWithdrawalRebalance()`

Severity: Medium	Status: Resolved

Context:

- StrategOperatorProxy.sol#L125-L135

Description: In `vaultWithdrawalRebalance()`, if the user uses `dynParamsExit`, the signature of the authority member needs to be provided.

```
    function vaultWithdrawalRebalance(
        address _user,
        address _vault,
        uint256 _deadline,
        uint256 _amount,
        bytes memory _signature,
        bytes memory _portalPayload,
        bytes memory _permitParams,
        uint256[] memory _dynParamsIndexExit,
        bytes[] memory _dynParamsExit
    ) external payable returns (uint256 returnedAssets) {
        if(msg.sender != addressProvider.userInteractions()) revert
OnlyUserInteraction();
        if(_deadline < block.timestamp) revert DeadlineExceeded();

        bool isProtected = _dynParamsIndexExit.length > 0;
        if(isProtected) {
            bytes32 signedHash = keccak256(
                abi.encode(
                    _user,
                    userWithdrawalRebalancedNonce[_user],
                    _deadline,
                    _dynParamsIndexExit,
                    _dynParamsExit
                )
            );
```

The problem here is that the signature does not include the `_vault` parameter, which means that the user can use that `dynParamsExit` in any vault. Considering the different strategies of vault, the same `dynParamsExit` may have side effects in different vaults.

Recommendation: It is recommended to include `_vault` in `signedHash`.

Strateg: The recommandations has been implemented:

- StrategOperatorProxy.sol#L125

Zenith: Verified.

[M-11] `putInBuffer()` doesn't work with rebase tokens.

Severity: Medium	Status: Resolved

Context:

- StrategAssetBuffer.sol#L22-L25

Description: The user can deploy the Vault with any tokens, and when the user makes a deposit into the Vault, `StrategAssetBuffer.putInBuffer()` is called to transfer the tokens into the buffer and approve the Vault for spending.

```
    function putInBuffer(address _asset, uint256 _amount) external {
        IERC20(_asset).safeTransferFrom(msg.sender, address(this),
_amount);
        IERC20(_asset).safeIncreaseAllowance(msg.sender, _amount);
    }
```

However, when the Vault's assets are rebase tokens, the amount approved will be different from the actual balance, resulting in the Vault being able to spend more or less tokens in the buffer.

Recommendation: It is recommended to apply whitelist restrictions to the assets available for Vault deployment, or just document this issue.

Strateg: This has already been whitelisted on the dApp and it will be added to our documentation

Zenith: Verified

[M-12] A malicious user can frontrun permit transaction to make it revert due to invalid signature

Severity: Medium	Status: Resolved

Context:

- LibPermit.sol#L56-L59
- LibPermit.sol#L27-L41
- StrategOperatorProxy.sol#L151-L152
- StrategUserInteractions.sol#L312-L321

Description: A malicious user can frontrun a permit transaction and directly use the signature exposed in the permit transaction. For `executePermit()`, when the malicious user uses the signature (call `asset.permit()` directly), the signature in `executePermit()` will be invalid and the transaction will revert.

```
    function executePermit(address _asset, address _from, address _to,
uint256 _amount, bytes memory _permitParams) internal {
        DataTypes.PermitParams memory p = abi.decode(_permitParams,
(DataTypes.PermitParams));
        ERC20Permit(_asset).permit(_from, _to, _amount, p.deadline, p.v,
p.r, p.s);
    }
```

Recommendation: It is recommended to wrap `asset.permit()` with try catch block in `executePermit()`.

Strateg: The recommendation has been implemented:

- LibPermit.sol#L58

Zenith: Verified.

4.3 Low Risk

A total of 10 low risk findings were identified.

[L-1] Missing array length checks in `StrategVaultFactory.editVaultParams()`

Severity: Low	Status: Resolved

Context: StrategVaultFactory.sol#L253-L279

Description:

The `StrategVaultFactory.editVaultParams()` function does not ensure that `_settings.length == _data.length`, so it is possible for both arrays to have different lengths.

The impact isn't really severe as:

- If `_settings.length > data.length`, `_editVaultParams()` reverts due to an out-of-bounds access on `_data`.
- If `_settings.length < data.length`, the remaining elements in `_data` are ignored.

In the worst-case scenario, the vault owner accidentally adds `_data` with less elements and his settings change can't be executed after going through the 3-day timelock as `executeVaultParamsEdit()` reverts. **Recommendation:**

Consider ensuring that the length of both arrays are the same:

```
if (_settings.length != _data.length) revert ArrayLengthsMismatch();
```

Strateg: Fixed with the following commit

Zenith: Verified.

[L-2] Missing `_vault` address validation in `StrategVaultFactory `functions

Severity: Low	Status: Resolved

Context: StrategVaultFactory.sol

Description:

In `StrategVaultFactory`, all functions going through `StrategUserInteractions` don't actually validate that `_vault` is a vault address deployed by this contract. This allows the user to specify `_vault` as any arbitrary address as long as the call passes.

There doesn't seem to be a way to exploit this in the current iteration of the code, but nevertheless, it is best to validate that the `_vault` address was deployed by the factory.

Recommendation:

Consider adding the following check to `setVaultStrat()`, `editVaultParams()`, `executeVaultParamsEdit()` and `vaultDeposit()`:

```
if (vaultConfiguration[_vault].erc3525 == address(0)) revert
InvalidVault();
```

`vaultConfiguration.erc3525` is only set for vaults that have been deployed through `deployNewVault()`, it is used to check if a vault is valid.

Strateg: The checks are done with the following modifier. It has been added on all functions that have a vault address in their inputs.

Zenith: Verified.

[L-3] `StrategVault.redeem()` can be forced to revert if the shares redeemed is close to `bufferShares`

Severity: Low	Status: Acknowledged

Context:

- StrategVault.sol#L335-L338

Description: The maximum amount of shares a user can withdraw through `redeem()` is determined in `maxRedeem()` as such:

```
uint256 assetsInBuffer = IERC20(asset()).allowance(buffer,
address(this));
uint256 bufferShares = convertToShares(assetsInBuffer);
uint256 ownerBal = balanceOf(__owner);
return bufferShares < balanceOf(__owner) ? bufferShares : ownerBal;
```

As seen from above, the maximum amount of shares redeemable is limited by `bufferShares`, which is the share equivalent of the total assets held in the buffer.

However, if a user tries to redeem an amount of shares close to `bufferShares`, it's possible to make the call to `redeem()` revert by donating assets and inflating the share price.

For example:

- Assume the vault has the following state:
  - `totalSupply() = 2e18`, `totalAssets() = 2e18`
  - The buffer holds `1e18` assets.
- User calls `redeem()` with `_shares = 1e18` to withdraw `1e18` assets.
- Attacker front-runs the user and transfers 1 wei of assets to the vault contract.
- When the call to `redeem()` is executed, `bufferShares` is calculated as:

```
assetsInBuffer * totalShares / totalAssets = 1e18 * 2e18 / (2e18 + 1) =
1e18 - 1
```

- Since `bufferShares` is smaller than `1e18`, `redeem()` reverts.

Recommendation: The user can call redeem() to retry their withdrawal, so this issue doesn't need fixing. Documenting this limitation may help avoid problems with external integrations.

Strateg: Acknowledged

[L-4] Exit hook is not called when performing a withdrawal in `StrategVault.withdrawalRebalance()`

Severity: Low	Status: Resolved

Context:

- StrategVault.sol#L284-L290

Description:

When a withdrawal is performed in `withdraw()` and `redeem()`, exit hooks are executed for all strategies before the withdrawal by calling `_executeHooks(DataTypes.BlockExecutionType.EXIT)`:

```
_executeHooks(DataTypes.BlockExecutionType.EXIT);

// ...

_withdraw(sender, _receiver, __owner, _assets, shares);
```

However, even though `withdrawalRebalance()` also performs a withdrawal, exit hooks are not executed in the function. This is inconsistent with the other withdrawal functions and might break composability with strategies.

Recommendation:

Call `_executeHooks(DataTypes.BlockExecutionType.EXIT)` in `withdrawalRebalance()` before performing asset calculations and performing the withdrawal.

Strateg: The hook execution has been added

Zenith: Verified.

[L-5] `minUserDeposit` limit in `StrategVault` can be bypassed

Severity: Low	Status: Resolved

Context:

- StrategVault.sol#L920-L930
- StrategVault.sol#L276-L293

Description:

Whenever users call `deposit()` or `mint()` to deposit assets into a `StrategVault`, if the `minUserDeposit` configuration parameter is set, a minimum deposit limit is enforced on the total amount of deposits they have:

```
//Minimum limit set
if (mode % 2 == 1) {
    uint256 minDeposit = minUserDeposit;
    if (userDeposited == 0) {
        if (_amount < minDeposit) revert MinDepositNotReached();
    } else {
        if (userDeposited + _amount < minDeposit) {
            revert MinDepositNotReached();
        }
    }
}
```

However, since `_applyTvlLimits()` is not called in `withdraw()`, `redeem()` or `withdrawalRebalance()`, this limit is not enforced when a withdrawal is performed, Therefore, it's possible for a depositor to bypass the minimum deposit limit by depositing followed by withdrawing immediately afterwards.

Recommendation:

In the `_update()` hook, consider checking that the user's new share balance is not below `minUserDeposit` when shares are burnt.

Strateg: A check on the sender balance has been added on the _applyTvlLimits function. Here, if the remaining balance of a user is greater than 0 after a withdrawal or a transfer, it verifies if the balance of the sender stays greater than the minimum limit

Zenith: Verified.

[L-6] `StrategVault#_rebalance` fails to clear highly sensitive dynamic parameters when exiting to buffer

Severity: Low	Status: Resolved

Context

StrategVault.sol#L685-L728

Description

Dynamic parameters are used to communicate important parameters when interacting with the block contracts.

StrategVault.sol#L719-L728

```
        if (currentBufferSize < bufferSize - derivation) {
            _storeDynamicParams(_dynParamsIndexExit, _dynParamsExit);
            _exitStrategy(bufferSize - currentBufferSize);
            _sendVaultAssetsInBuffer();
        }

        isLive = true;

        _purgeDynamicParams(_dynParamsIndexEnter);
    }
```

As seen above, dynamic parameters are stored when exiting the due to undersized buffer but they are never purged from storage afterwards. This leaves highly sensitive parameters in storage that can be accessed outside of the intended scope of the transaction.

Recommendation

Purge dynamic exit parameters as well.

Strateg: This function has been changed, now there is only 2 parameters as inputs of the function. It made no sense to have two times dynamic params because if dynamic params for entering the strategy were called the two other was never used and vice versa. So they have been regrouped in the same 2 variables - here.

Zenith: Verified - Contract now utilizes a single set of inputs parameters and correctly clears them after usage.

[L-7] StrategVault#withdraw and redeem applies timelock to sender which can lead to incorrectly reverting transactions

Severity: Low	Status: Resolved

Context

StrategVault.sol#L276-L293

Description

StrategVault.sol#L276-L293

```
function withdraw(uint256 _assets, address _receiver, address __owner)
    public
    virtual
    override
    nonReentrant
    returns (uint256)
{
    address sender = _msgSender();
    _executeHooks(DataTypes.BlockExecutionType.EXIT);
    _applyTimelock(sender);
    if (_assets > maxWithdraw(__owner)) revert WithdrawMoreThanMax();

    uint256 shares = previewWithdraw(_assets);
    _decreaseValueDeposited(sender, shares);
    _withdraw(sender, _receiver, __owner, _assets, shares);

    return shares;
}
```

When withdrawing and redeeming, `_applyTimelock` is called for `sender` rather than `owner`. Since owner is the one providing the shares, it would be incorrect to revert if `sender` was timelocked.

Recommendation

As discussed, this should be removed and all validation should happen in `_update`

Strateg: All validation has been moved to _update function

Zenith: Verified.

[L-8] _rebalance does not consider assets in the vault

Severity: Low	Status: Resolved

Context:

- StrategVault.sol#L460-L467

Description:

When rebalance is called, it does not first transfer the assets in the vault to the buffer, which results in the actual bufferSize being larger than asset.allowance(), which could cause rebalance to remove an incorrect amount of assets from the strategy.

```
    function rebalance(
        uint256[] memory _dynParamsIndexEnter,
        bytes[] memory _dynParamsEnter,
        uint256[] memory _dynParamsIndexExit,
        bytes[] memory _dynParamsExit
    ) external onlyOperator {
        _rebalance(_dynParamsIndexEnter, _dynParamsEnter,
_dynParamsIndexExit, _dynParamsExit);
    }
```

Vault may retain assets for the following reasons.

1. when `strategyBlocksLength` is 0, `_enterInStrategy()` does not deposit assets from the vault into the strategy.
2. `withdrawalRebalance()` withdraws assets greater than `totalWithdrawal`.

Recommendation:

It is recommended to call `_sendVaultAssetsInBuffer()` before calling `_rebalance()`.

Strateg:

In the new implementation, a check has been added to avoid strategyBlocksLength = 0 StrategVaultFactory.sol#L225

and the rebalance is calculated from the _getNativeTvl and the buffer allowance. the _getNativeTvl taking care of the assets owned by the vault. StrategVault.sol#L795

Zenith: Verified

[L-9] The pending settings cannot be canceled.

Severity: Low	Status: Resolved

Context:

- StrategVaultFactory.sol#L253-L279

Description:

When the owner edits the vault's settings, the settings are pushed to the `vaultParamsChangeQueue`, and the settings are applied three days later.

```
        } else {
            uint256 index = vaultParamsChangeQueue[_vault].length;
            VaultParametersEditQueueItem memory changes =
VaultParametersEditQueueItem({
                initializedAt: block.timestamp,
                settings: _settings,
                settingsData: _data
            });
            vaultParamsChangeQueue[_vault].push(changes);

            emit NewVaultParametersEditQueueItem(_vault, index);
        }
```

The problem here is that once the settings are pushed, the owner cannot cancel them, the setting will always be applied.

Additionally, when there are multiple settings pushed to the queue, there's nothing in `executeVaultParamsEdit()` that ensures changes are executed in the order they were added.

This could create scenarios where settings end up different to what the owner expects, for example:

- Owner calls `editVaultParams()` to set `timelockDuration` to 50 days. We call this change A.
- Owner changes his mind, he calls `editVaultParams()` to change `timelockDuration` to 100 days instead. We call this change B.
- Change A passes the timelock of `EDIT_SETTINGS_QUEUE_DURATION`, but isn't executed.
- After some time, change B passes the timelock as well.
- Attacker calls `executeVaultParamsEdit()` twice to execute change B, then change A afterwards.

- `timelockDuration` is set to 50 days, instead of 100 days as the owner intended.

Essentially, when there is more than one queued change, the owner has to monitor and execute the changes as they pass the timelock, otherwise an attacker could execute them in a different sequence.

Recommendation:

It is recommended to implement a function to allow the owner to cancel the settings.

Allowing the owner to cancel changes would mean the owner doesn't have to pay attention to when the queued changes become executable - in the scenario described above, the owner can just cancel change A.

Strateg:

A new queue system has been implemented for vault settings change. This implementation ensures:

- The execution of changes in the right order.
- Allow to cancel a queued change.
- StrategVaultFactory.sol#L318

[L-10] `removeTokenPercent()` and `removeAllTokenPercent()` do not work as described

Severity: Low	Status: Resolved

Context:

- LibOracleState.sol#L111-L128

Description:
The comments for `removeTokenPercent()` and `removeAllTokenPercent()` indicate that `_percent` is the percentage to be reduced, but actually it's the percentage that's to be left.

```
    /**
     * @notice Applies a percentage reduction to a specific token's
amount in the oracle state.
     * @param self The oracle state to modify.
     * @param _token The token address to apply the reduction to.
     * @param _percent The percentage (in basis points) to reduce the
token's amount by.
     */
    function removeTokenPercent(DataTypes.OracleState memory self,
address _token, uint256 _percent) internal pure {
        for (uint256 i = 0; i < self.tokens.length; i++) {
            if (self.tokens[i] == _token) {
                self.tokensAmount[i] = (self.tokensAmount[i] * _percent)
/ 10000;
            }
        }
    }
```

Recommendation: It is recommended to change as follows

```
    function removeTokenPercent(DataTypes.OracleState memory self,
address _token, uint256 _percent) internal pure {
        for (uint256 i = 0; i < self.tokens.length; i++) {
            if (self.tokens[i] == _token) {
-               self.tokensAmount[i] = (self.tokensAmount[i] * _percent)
/ 10000;
+               self.tokensAmount[i] -= (self.tokensAmount[i] * _percent)
/ 10000;
            }
        }
```

```
    }
...
    function removeAllTokenPercent(DataTypes.OracleState memory self,
uint256 _percent) internal pure {
        for (uint256 i = 0; i < self.tokens.length; i++) {
-           self.tokensAmount[i] = (self.tokensAmount[i] * _percent) /
10000;
+           self.tokensAmount[i] = (self.tokensAmount[i] * _percent) /
10000;
        }
    }
```

Strateg: This changes has been applied:

- LibOracleState.sol#L126

Zenith: Verified.

4.4 Informational

A total of 3 informational findings were identified.

[I-1] Specifying `gasleft()` as the gas limit in `StrategVaultFactory.vaultDeposit()` is redundant

Severity: Informational	Status: Resolved

Context: StrategVaultFactory.sol#L327-L328

Description:

In `StrategVaultFactory.vaultDeposit()`, `deposit()` is called while specifying the gas limit as `gasleft()`:

```
uint256 gas = gasleft();
(bool success, bytes memory returndata) = _vault.call{gas: gas}(
```

However, specifying `gasleft()` as the gas limit for the `deposit()` here is redundant, the low-level call will never send more gas than the remaining amount.

Recommendation:

Consider performing the low-level call normally:

```
- uint256 gas = gasleft();
- (bool success, bytes memory returndata) = _vault.call{gas: gas}(
+ (bool success, bytes memory returndata) = _vault.call(
```

Strateg: Redundant check removed on the following line.

Zenith: Verified.

[I-2] Missing length checks on `_dynParamsIndex` and `_dynParams` in `StrategVault._storeDynamicParams()`

Severity: Informational	Status: Resolved

Context: StrategVault.sol#L735-L740

Description:

In the `StrategVault` contract, when the operator calls functions that require entering or exiting strategies, they specify `_dynParamsIndex` and `_dynParams`, which are the dynamic parameters for each individual strategy.

However, these functions do not ensure that `_dynParamsIndex` and `_dynParams` have the same lengths. In the event that the operator misses out on one parameter, it might cause unexpected behavior when entering/exiting strategies.

Recommendation:

In `_storeDynamicParams()`, consider ensuring that the lengths of `_dynParamsIndex` and `_dynParams` are equal:

```
  uint256 arrLength = _dynParamsIndex.length;
+ if (arrLength != _dynParams.length) revert UnequalLengths();
  for (uint256 i = 0; i < arrLength; i++) {
      LibBlock.setupDynamicBlockData(_dynParamsIndex[i], _dynParams[i]);
  }
```

Strateg: The length checks has been added here.

Zenith: Verified.

[I-3] Calls to `maxMint` and `maxDeposit` as validation during deposits and mints are unnecessary

Severity: Informational	Status: Resolved

Context

- StrategVault.sol#L221
- StrategVault.sol#L251

Description

StrategERC4626Upgradeable.sol#L167-L169

```
function maxDeposit(address) public view virtual returns (uint256) {
    return type(uint256).max;
}
```

StrategERC4626Upgradeable.sol#L174-L176

```
function maxMint(address) public view virtual returns (uint256) {
    return type(uint256).max;
}
```

During deposits and mints, `maxDeposit` and `maxMint` are called to validate that the deposit or mint is valid. However these functions always return `uint256.max` which is a meaningless check.

Recommendation

Remove these checks for mints and deposits.

Strateg: This checks has been removed here and here.

Zenith: Verified.