yAcademy Yearn Stargate Strategy review

Review Resources:

- Stargate User Docs
- Stargate Developer Docs
- Recent bug in Stargate strategy accounting

Residents:

- engn33r

Fellows:

- spalen
- shung
- hasanza
- DecorativePineapple

Table of Contents

12 About yAcademy

Review Summary

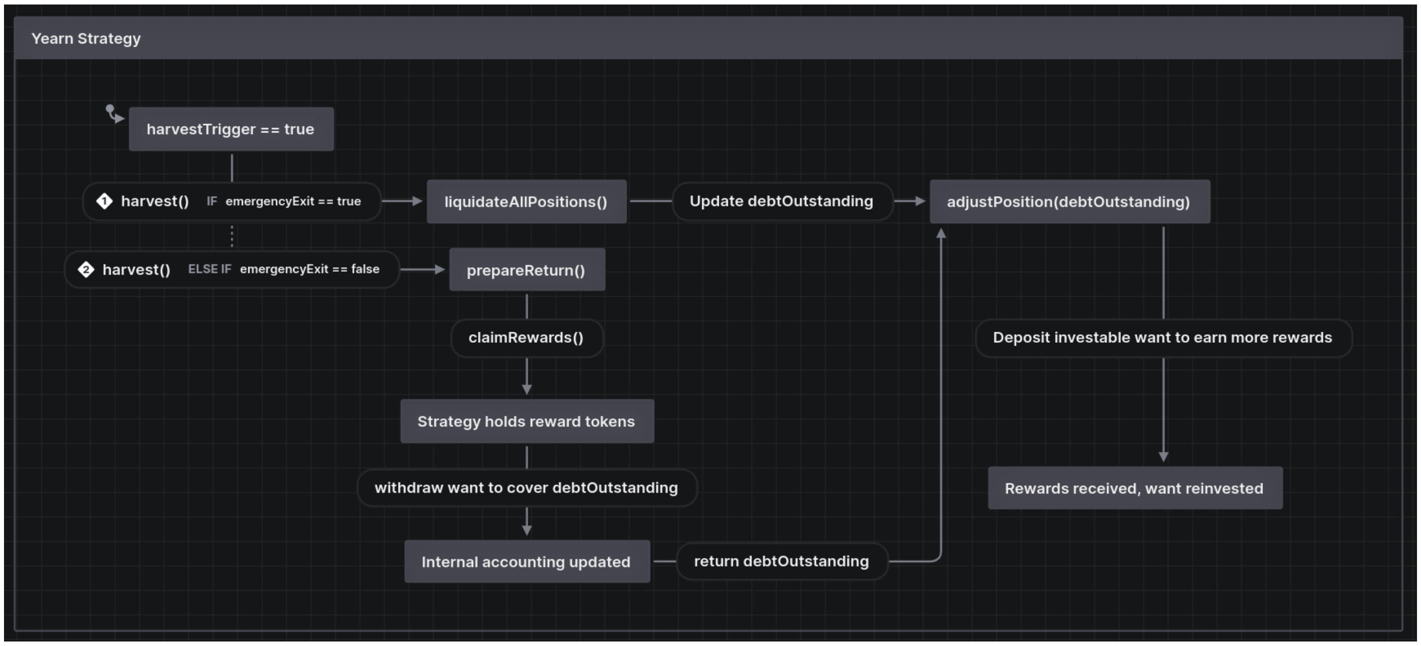
Yearn Stargate Strategy

The Yearn Finance Stargate Strategy generates yield from the Stargate protocol. The strategy is currently designed for USDC and USDT on Ethereum Mainnet but the strategy may be deployed to other chains, such as Optimism or Arbitrum, in the future. The yield is generated in two ways: from STG staking rewards (using a modified MasterChef contract) and from a portion of the 0.06% bridge fee collected by Stargate that is redistributed back to the liquidity pools.

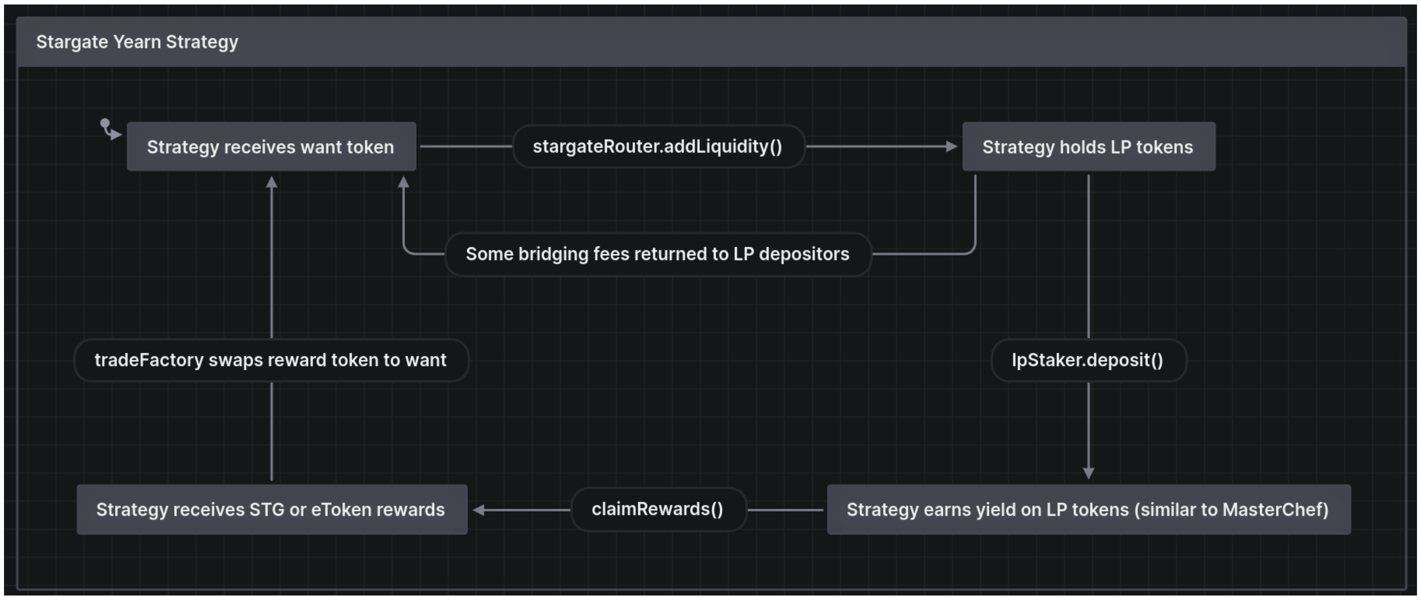
The contracts of the Yearn Finance Stargate Strategy Repo were reviewed over 14 days, 2 of which were used to create an initial overview of the contract. The code review was performed between November 20 and December 4, 2022. The code was reviewed by 1 resident for a total of 28 review hours (engn33r 28 hours). A number of yAcademy Fellows from Fellowship block 4 also reviewed the contracts and contributed over 40 man hours. The repository was under active development during the review, but the review was limited to the latest commit at the start of the review. This was commit bcbc1ddbe8973b55a6178f5e07177332938f0cd2 for the Yearn Finance Stargate Strategy repo.

Scope

The scope of the review consisted of the entire Yearn Finance Stargate Strategy repo at the specific commit, but the core logic was located in the Strategy.sol contract. This strategy was already deployed to mainnet in the form of a USDC strategy and a USDT strategy with millions of dollars of TVL. The strategy follows the standard format of a Yearn strategy, where the underlying token is deposited into the Yearn vault, and the vault distributes the token to strategies that invest the token in different ways to earn yield.


Yearn Strategy
harvestTrigger == true
harvest() IF emergencyExit == true
liquidateAllPositions()
Update debtOutstanding
adjustPosition(debtOutstanding)
harvest() ELSE IF emergencyExit == false
prepareReturn()
claimRewards()
Strategy holds reward tokens
withdraw want to cover debtOutstanding
Internal accounting updated
return debtOutstanding
Deposit investable want to earn more rewards
Rewards received, want reinvested

This specific strategy earns yield with Stargate, a bridge that describes itself as solving the bridging trilemma. The features the Stargate bridge offers include: instant guaranteed finality, unified liquidity in a single pool to improve capital efficiency and avoid fractured liquidity, and the ability to use native assets without wrapping the assets. The yield is earned from bridge fees and from LP staking rewards.


Stargate Yearn Strategy
Strategy receives want token
stargateRouter.addLiquidity()
Strategy holds LP tokens
Some bridging fees returned to LP depositors
tradeFactory swaps reward token to want
lpStaker.deposit()
Strategy receives STG or eToken rewards
claimRewards()
Strategy earns yield on LP tokens (similar to MasterChef)

The Yearn Stargate strategy is only deployed on Ethereum mainnet for USDC and USDT at present, but the strategy may be deployed to Optimism and/or Arbitrum in the future. After the findings were presented to the Yearn strategists team, fixes were made and included in several PRs.

This review is a code review to identify potential vulnerabilities in the code. The reviewers did not investigate security practices or operational security and assumed that privileged accounts could be trusted. The reviewers did not evaluate the security of the code relative to a standard or specification. The review may not have identified all potential attack vectors or areas of vulnerability.

yAcademy and the residents make no warranties regarding the security of the code and do not warrant that the code is free from defects. yAcademy and the residents do not represent nor imply to third parties that the code has been audited nor that the code is free from defects. By deploying or using the code, the Yearn Stargate Strategy and users of the contracts agree to use the code at their own risk.

Code Evaluation Matrix

Category	Mark	Description
Access Control	Good	Yearn strategies follow a pattern that enforces good separation between user-controller functions in the vault and internal mechanics of the strategies that the keepers and governance control.
Mathematics	Average	The internal accounting in `prepareReturn()` contains some complexity around converting between LP units and want tokens, but this review determined that the accounting has no major errors.
Complexity	Average	This Yearn strategy has average complexity compared to other Yearn strategies because it follows a similar deposit/withdraw pattern and does internal accounting to convert between want and LP shares.
Libraries	Average	This strategy has similar imports to other strategies, though importing the Stargate interface files is the main novelty of this strategy.
Decentralization	Average	This strategy is managed in the same way as other Yearn strategies, with the multisig handling overall management of Yearn Finance.

Category	Mark	Description
Code stability	Good	This strategy was already holding value on mainnet, which indicates that the code is production-ready.
Documentation	Average	The documentation for this strategy was average, with inline comments to indicate what the internal account logic is trying to achieve. But the Stargate protocol which this strategy deposits to does not have the best documentation. Even the Stargate whitepaper that explains the delta credit algorithm does not offer a clear summary of all the mechanics.
Monitoring	Good	Yearn strategies have solid monitoring in place thanks to yearn.vision and yearn.watch. Events are handled by the inherited BaseStrategy contract.
Testing and verification	Good	The strategy was already deployed to mainnet and therefore has had substantial testing, from developers running tests and from real users.

Findings Explanation

Findings are broken down into sections by their respective impact:

- Critical, High, Medium, Low impact
  - These are findings that range from attacks that may cause loss of funds, impact control/ownership of the contracts, or cause any unintended consequences/actions that are outside the scope of the requirements
- Gas savings
  - Findings that can improve the gas efficiency of the contracts
- Informational
  - Findings including recommendations and best practices

Critical Findings

None.

High Findings

None.

Medium Findings

None.

Low Findings

1. Low - Discrepancy in BaseStrategy versioning (engn33r)

The mainnet Stargate strategy contracts have a `apiVersion` of v0.4.3. At the same time, the strategy features a `baseFeeOracle` address and a `isBaseFeeAcceptable` check in `harvestTrigger()`, and these baseFee values were only introduced in v0.4.5. It is unclear whether the strategy is supposed to use v0.4.5 of the BaseStrategy or is in the process of migrating from v0.4.3 to v0.4.5 and has not fully made the upgrade.

Technical Details

The git blame of BaseFeeOracle.sol from the yearn-vaults repo shows that the entire file is a new addition for yearn-vaults release v0.4.5. The file diff for PR #546 confirms the introduction of baseFee values in the v0.4.5 release.

Impact

Low. The specifics of interactions with BaseStrategy variables and functions of different versions were out of scope for this review, but a strategy should comply with a single BaseStrategy version (instead of merging elements of different versions) for maximum compatibility with the Yearn ecosystem.

Recommendation

Fully upgrade the strategy to inherit BaseStrategy v0.4.5 instead of the currently used v0.4.3. Make additional changes as necessary to move away from the imported 0.4.3 BaseStrategy contract.

Developer Response (Val John)

Acknowledged. The upcoming revision of the Stargate strategy is meant for vault 0.4.5, any discrepancies related to earlier vault version will be removed.

2. Low - `estimatedTotalAssets()` does not include value of reward tokens (engn33r)

The LPStaking contract (and LPStakingTime contract on Optimism) returns STG (or eToken on Optimism) to the staker. The value of these rewards is not accounted for in `estimatedTotalAssets()`.

Technical Details

The function name `estimatedTotalAssets()` implies all value held by the strategy is accounted for. The rewards from the staking pool are not, however, included in this calculation. This means after `_claimRewards()` is called in `prepareReturn()`, the value of `_profit` doesn't include the rewards just claimed. The `_profit` only includes these rewards after ySwap or Cowswap swaps the `reward` to `want`, which causes a delay in perceived profits.

It may be intentional to only include the value of want and LP tokens in this calculation. Because this is a public function, other users or protocols may assume the function does account for all forms of value held by the strategy, which is an incorrect assumption. Instead, after rewards are claimed by the strategy and are converted into want tokens, the total assets of this strategy will suddenly increase. The amount of this sudden increase could vary depending on how often the rewards are redeemed from the staking pool and how often they are swapped for want.

Beyond the inclusion of rewards in the profit and asset values, without guarantees around when the ySwap process will happen, it is possible that the strategy may hold the reward tokens in the form of STG or eTokens for some time. This would delay the start of compound interest on those rewards. Clearer guarantees or incentives to minimize the time between `harvest()` and the ySwap would resolve this.

Impact

Low. External integrations may not understand that the values returned by functions like `estimatedTotalAssets()` do not include the rewards that have not been swapped for `want` yet.

Recommendation

Consider including the value of rewards in the calculation of `estimatedTotalAssets()` or enable the ySwap soon after a harvest to allow for proper updating of relevant values.

Developer Response (Val John)

The `estimatedTotalAssets()` purposefully omits the reward token (unclaimed and claimed) due to the intended selling mechanism. Under this strategy, the rewards are first claimed then sold async via tradeFactory. Rewards cannot be sold atomically due to the nature of the cowswap solver. This result in profit from reward claimed accounted for in the next harvest only.

3. Low - Loss calculation discrepancies (engn33r)

The loss calculation in `prepareReturn()` and `withdrawSome()` can be inaccurate in some edge cases. These edge cases are:

1. If the Stargate protocol is hacked or has technical problems with delta credit calculations that result in LP tokens becoming impossible to withdraw
2. When there is a non-negligible amount of value in reward tokens (STG tokens on mainnet, eTokens on Optimism)

Technical Details

The loss calculation in `prepareReturn()` is calculated with several assumptions. We will examine each step of the calculations and the relevant assumptions.

1. The first loss calculation is in `withdrawSome()`. The loss will be zero if `_potentialLoss < balanceOfLPTokens`. This calculation assumes the LP tokens have the value that the Stargate pool advertises, which is a multiple of the underlying want asset (with bridge fees slowly adding to this value). But relying on the Pool to provide the value of an LP token assumes that the Pool is working normally and all LP tokens can be withdrawn. If the LP tokens cannot be withdraw, they will have zero value and cannot be assumed to compensate for any potential loss. The delta credit calculation is a complex algorithm, so it should be evaluated whether this complexity may introduce risk. The loss calculation may be problematic for `liquidatePosition()`, where the BaseStrategy describes the return values with the comment `This function should return the amount of want tokens made available by the liquidation. If there is a difference between them, _loss indicates whether the difference is due to a realized loss, or if there is some other sitution at play (e.g. locked funds) where the amount made available is less than what is needed.` Because `withdrawSome()` returns a loss of 0 even when there are locked funds, it may not meet with the described goal of the loss return value in `liquidatePosition()` if locked funds are considered losses (either because of temporary delta credit limits or because of a hack).

2. The second loss calculation is in `prepareReturn()`. The calculation depends on whether `_vaultDebt > _totalAssets` is true. The problem with this is that a separate finding explains how `estimatedTotalAssets()` does not include the value of reward tokens, and `_totalAssets = estimatedTotalAssets()` in this calculation. If there is substantial value stored in reward tokens, the value of `_totalAssets` would be lower than the value of assets held by the strategy, meaning the loss value may be calculated as a larger loss than in reality once the reward tokens are factored in.
3. The third loss calculation is at the end of `prepareReturn()`. This calculation compares loss to profit, but like the previous step of loss calculations, profit is dependent on `estimatedTotalAssets()`, so the profit may be less than in actuality. If `_loss > _profit`, then the reduction in loss may partially cancel out the inflation of the loss from the previous step, but if `_loss <= _profit`, then loss would not receive this adjustment.

Impact

Low. The inaccuracy in loss calculations are most likely minor, depending on how the vault uses this value in later steps. This issue should not be a major concern if the ySwap happens often and the Stargate protocol has no technical issues or security issues.

Recommendation

Consider including the value of rewards in the calculation of `estimatedTotalAssets()` or enable the ySwap soon after a harvest to allow for proper updating of relevant values. Separately, if there is a method of estimating the market value of Stargate LP tokens that does not rely on the Stargate pool, this could provide a sense check in the event that tokens are locked in the Stargate pool. If needed, consider adjusting the loss value in `wantSome()` or `liquidatePosition()` to meet the goals of the BaseStrategy `liquidatePosition()` return values.

Developer Response (Val John)

Acknowledged. The two (2) edge cases highlighted are valid. However, to our knowledge there is no other venue that could be used to estimate the market value of Stargate LP token.

4. Low - Variables set to Ethereum addresses (spalen)

At L99 `healthCheck` and L100 `baseFeeOracle` variable are set to specific addresses that existing on Ethereum.

Technical Details

The strategy is planned to use on other chains, like Optimism. These variables should be defined per chain.

Impact

Low. There is a possibility to deploy contracts on these addresses and provide invalid data and manipulate harvest functionality.

Recommendation

Don’t set values to these variables explicitly. Use set functions to define these variables per chain.

Developer Response (Val John)

Acknowledged. This is a good point and these addresses will be removed in effort to standardize the strategy.

5. Low - Emergency unstake forgoes extra rewards even when they can be claimed (shung)

`_emergencyUnstakeLP` function forgoes pending rewards even if the rewards can be harvested safely. This will result in migration to always lose pending rewards.

Technical Details

`_emergencyUnstakeLP` is an internal function executed during migration, or during admin-initiated emergency exits. In the former case, migration can be required even if the staker contract is still functional and is receiving rewards. In the latter case, admin can call emergency unstake by mistake when the rewards are still claimable. In both cases, pending rewards will be discarded and irrecoverably stuck in the Stargate LP staker contract.

Impact

Low. Loss of pending rewards unless admin ensures the pending rewards are harvested right before migration or emergency exit.

Recommendation

Consider using try-catch statement to first try harvesting the rewards. This way, if the rewards are harvestable, they will be harvested; if they are not harvestable, the function will not revert at harvest attempt and still execute the emergency withdraw. Below is a diff to demonstrate the suggested fix.

```diff
diff --git a/contracts/Strategy.sol b/contracts/Strategy.sol
index 461383c..13d4c8b 100644
--- a/contracts/Strategy.sol
+++ b/contracts/Strategy.sol
@@ -466,6 +466,7 @@ contract Strategy is BaseStrategy {
     }

     function _emergencyUnstakeLP() internal {
+        try lpStaker.deposit(liquidityPoolIDInLPStaking, 0) {} catch {}
         lpStaker.emergencyWithdraw(liquidityPoolIDInLPStaking);
     }
```

Developer Response (Val John)

Acknowledged. Strategy amended per recommendations.

Gas Savings Findings

1. Gas - Duplicate line of code (engn33r)

The `prepareReturn()` function in the strategy sets `forceHarvestTriggerOnce` to false before the function ends. But the BaseStrategy performs the same action immediately after the line in which `prepareReturn()` is called.

Technical Details

In `prepareReturn()`, the `forceHarvestTriggerOnce` variable gets set to false. However, the BaseStrategy does the exact same thing right after the call to `prepareReturn()` is completed, even with the same comment. This line of code (and the comment) is redundant.

Impact

Gas savings.

Recommendation

Remove the duplicate line of code in `prepareReturn()` to allow the BaseStrategy to set this variable.

2. Gas - Use BaseStrategy logic for gas savings (engn33r, shung)

The BaseStrategy code in `harvestTrigger()` uses one less if statement than the similar implementation in the Stargate Strategy.

Technical Details

The BaseStrategy saves an if statement in `harvestTrigger()` by returning the final boolean directly. The current strategy implementation is not as direct.

Impact

Gas savings.

Recommendation

Edit the return value to use the same final return value as BaseStrategy uses:

```
-        if (vault.creditAvailable() > creditThreshold) {
-            return true;
-        }
-
-        // otherwise, we don't harvest
-        return false;
+        return vault.creditAvailable() > creditThreshold;
```

3. Gas - Use Solidity errors in 0.8.4+ (engn33r)

Using solidity errors is a new and more gas efficient way to revert on failure states.

Technical Details

Require statements are used in the strategy and error messages are not used anywhere. Using this new solidity feature can provide gas savings on revert conditions.

Impact

Gas savings.

Recommendation

Add errors to replace each `require()` with `revert errorName()` for greater gas efficiency.

4. Gas - Use simple comparison (engn33r)

Using a compound comparison such as ≥ or ≤ uses more gas than a simple comparison check like >, <, or ==. Compound comparison operators can be replaced with simple ones for gas savings.

Technical Details

The `prepareReturn()` function contains this code:

```
if (_liquidWant <= _profit) {
    _profit = _liquidWant;
    _debtPayment = 0;
// enough to pay for all profit and _debtOutstanding (partial or full)
} else {
    _debtPayment = Math.min(_liquidWant - _profit, _debtOutstanding);
}
```

By switching around the if/else clauses, we can replace the compound operator with a simple one

```
if (_liquidWant > _profit) {
    _debtPayment = Math.min(_liquidWant - _profit, _debtOutstanding);
} else {
    _profit = _liquidWant;
    _debtPayment = 0;
// enough to pay for all profit and _debtOutstanding (partial or full)
}
```

Impact

Gas savings.

Recommendation

Replace compound comparison operators with simple ones for gas savings.

5. Gas - Use unchecked if no underflow risk (engn33r, shung)

There is a subtraction operation that can use unchecked for gas savings.

Technical Details

The first location where unchecked can be applied

```diff
- _profit = _totalAssets > _vaultDebt ? _totalAssets - _vaultDebt : 0;
+ unchecked { _profit = _totalAssets > _vaultDebt ? _totalAssets - _vaultDebt : 0; }
```

The second location where unchecked can be applied

```diff
- (_amountFreed, _loss) = withdrawSome(
-                 _toLiquidate - _wantBalance
-             );
+ unchecked { (_amountFreed, _loss) = withdrawSome(
+                 _toLiquidate - _wantBalance
+             ); }
```

Similar savings can be found throughout the contract because many logic checks remain from a recent commit where the strategy was migrated from an older version of solidity that did not have built-in SafeMath to a newer version that does have SafeMath. Other examples are found here, here, here, here, here, here, here, here, here, and here.

Impact

Gas savings.

Recommendation

Use unchecked block if there is no overflow or underflow risk for gas savings.

6. Gas - Remove unneeded variables (engn33r, spalen)

1. `_amountFreed` is set but not used, making it unnecessary.
2. Another variable, the return variable `_liquidatedAmount` in `withdrawSome()`, can be replaced by the `_liquidAssets` variable.
3. The variable `_liquidWant` in `prepareReturn()` can be replaced by `_wantBalance`.

Technical Details

1. _amountFreed is set in `prepareReturn()` but is never used after that point. The variable and the lines setting it can be deleted to save gas.

2. In `withdrawSome()`, `_liquidatedAmount` is used only to return the proper value, and in on branch of the if statement `_liquidatedAmount` is set to `_liquidAssets`. Instead of

creating a new temporary variable `_liquidAssets`, use `_liquidatedAmount` instead and change the if statement logic to

```diff
- uint256 _liquidAssets = balanceOfWant() - _preWithdrawWant;
+ _liquidatedAmount = balanceOfWant() - _preWithdrawWant;
- if (_amountNeeded > _liquidAssets) {
+ if (_amountNeeded > _liquidatedAmount) {
-       _liquidatedAmount = _liquidAssets;
    uint256 balanceOfLPTokens = _lpToLd(balanceOfAllLPToken());
-       uint256 _potentialLoss = _amountNeeded - _liquidAssets;
+       uint256 _potentialLoss = _amountNeeded - _liquidatedAmount;
    _loss = _potentialLoss > balanceOfLPTokens ? _potentialLoss - balanceOfLPTokens:0;
} else {
    _liquidatedAmount = _amountNeeded;
}
```

1. At L218 there is no need define `_liquidWant` and call `balanceOfWant()`. Recalculate `_wantBalance` after L213. Remove variable `_liquidWant` and use `_wantBalance` instead.

Impact

Gas savings.

Recommendation

Remove or replace unnecessary variables.

7. Gas - Remove unneeded if statement (engn33r)

`withdrawSome()` checks the `_amountNeeded` function argument to see `if (_amountNeeded > 0)`. `withdrawSome()` is an internal function that is only called in two places. In both instances, the `_amountNeeded` function argument passed to `withdrawSome()` is greater than zero, making this if statement unnecessary because it will always evaluate to true.

Technical Details

`withdrawSome()` contains an if statement of `if (_amountNeeded > 0)`. The two instances where this internal function is called is on line 210 and line 298. In both instances, the `_amountNeeded` function argument is greater than zero, so the if statement can be removed because the code in the if statement will always be used.

Impact

Gas savings.

Recommendation

Remove unnecessary if statement for gas savings. If this change is made, be aware that it could impact future changes of the strategy if `withdrawSome()` is called under different circumstances.

8. Gas - Unnecessary internal function call (engn33r)

`_lpToLd()` is only called once and the function is one line. It can be replaced with the one line it contains. The same could be done with `_stakeLP()`.

Technical Details

`_lpToLd()` calls `liquidityPool.amountLPtoLD()`. `_lpToLd()` is only used once, in `withdrawSome()`. `liquidityPool.amountLPtoLD()` is already called directly in `valueOfLPTokens()`, so `_lpToLd()` may as well be removed and the single instance of it replaced with a direct call to the external function.

The same approach could be taken with `_stakeLP()` which calls the external `lpStaker.deposit()` function.

Impact

Gas savings.

Recommendation

Remove the `_lpToLd()` function and replace the one place where it is used with a direct call to `liquidityPool.amountLPtoLD()`, or at a minimum, modify the code to use either `_lpToLd()` or `liquidityPool.amountLPtoLD()` consistently.

9. Gas - `balanceOfReward()` can be external (engn33r)

The `balanceOfReward()` function should be declared external, not public, for gas savings.

Technical Details

If a function is only called externally and does not need to be called internally, it can be declared external for gas savings. Apply this to `balanceOfReward()`.

Impact

Gas savings.

Recommendation

Declare function external instead of public.

10. Gas - Faster return from function (spalen)

Immediate return from function `withdrawSome(uint256 _amountNeeded)` on 0 input param.

Technical Details

If the input param `_amountNeeded` is 0 there is no need to do any calculations.

Impact

Gas savings.

Recommendation

Before L265 exit for 0 input value. After this check, if statement at L266 can be removed.

```
    if (_amountNeeded == 0) {
        return (0, 0);
    }
```

Informational Findings

1. Informational - Minor fix not applied to mainnet strats (engn33r)

The latest version of the strategy contract in the github repo has not been deployed to mainnet yet. The contracts live on mainnet do not use some of the changes made in recent commits.

Technical Details

Changes in recent commits, such as 36ddf75 and a1aaf31, contain changes that do not appear in the mainnet strategy contract code. The code on mainnet is using an older version of the contract code. Although the differences do not appear to impact the logic of the strategy, it is normally a good idea to update code on mainnet when improvements are made.

Impact

Informational.

Recommendation

Update the mainnet strategies to use the latest code upgrades when the latest changes have been reviewed sufficiently.

2. Informational - Incorrect comment(s) (engn33r)

A comment about `totalDebt` in the strategy is incorrect.

Technical Details

There is a comment `Amount of tokens that all strategies have borrowed` in the strategy on the line of code for `vault.strategies(address(this)).totalDebt`. This comment was copied from the Vault contract's comment about the Vault's `totalDebt` state variable, and does not properly describe the code on this line because this line uses the `totalDebt` value in the `StrategyParams` struct for a specific strategy.

Impact

Informational.

Recommendation

Change the comment to `Amount of tokens that this strategy has borrowed`.

3. Informational - Interfaces are not cleanly defined (engn33r, shung)

Some shortcuts are taken to combine two interface files into one. This can cause confusion in cases where the address is cast with an interface type that does not contain the function defined in the interface file.

Technical Details

Several unexpected situations are found in the interface files:

1. The standard IERC20 interface file does not have a `token()` function, but the IERC20Metadata.sol file has a `token()` function. This is probably a mistake that should be fixed because `token()` should only be called on pool addresses and is defined in IPool. Instead, consider adding `name()` to this interface like the OpenZeppelin IERC20Metadata.sol file.
2. The ILPStaking interface contains `pendingStargate()` and `pendingEmissionToken()` functions. These functions are not found in the same Stargate contract. The interface combines functions from LPStaking and LPStakingTime to avoid creating a new interface file.
3. `decimals()` can be removed from IWETH.sol because it is not used in the strategy.

Impact

Informational.

Recommendation

At a minimum, remove `token()` from IERC20Metadata. It might be worthwhile to better to create a ILPStakingTime interface that is split from ILPStaking for a future deployment on Optimism.

4. Informational - `tradeFactory.disable()` never called (engn33r)

When a new tradeFactory is set, the `enable()` function is called, but `disable()` is not called when an old tradeFactory is removed.

Technical Details

`setTradeFactory()` is designed to migrate to a new ySwaps trade factories. `_removeTradeFactoryPermissions()` removes the approval of the old `tradeFactory` before a new one is approved. `_removeTradeFactoryPermissions()` does not call `tradeFactory.disable()`, but it might be preferable to do so and follow the design of how the TradeFactory contract was written.

Impact

Informational.

Recommendation

Add `disable()` to ITradeFactory and call `tradeFactory.disable(address(reward), address(want))` in `_removeTradeFactoryPermissions()`.

5. Informational - Could use Stargate RouterETH for WETH/SGETH conversion (engn33r)

The strategy implements a custom `_convertWETHtoSGETH()` function while an existing Stargate function serves the same purpose. Note that the Stargate strategy has not been used for WETH yet but could do so in the future.

Technical Details

The list of Stargate contracts deployed on mainnet shows two router contracts, one named Router and one named RouterETH. Strangely, the RouterEth contract does not exist in the main branch of Stargate's github repository. The code is verified on Etherscan and shows that RouterEth.sol integrates with the SGETH contract to support ETH deposits. The `addLiquidityETH()` function in RouterEth could serve the same purpose as the custom `_convertWETHtoSGETH()` found in the strategy. Whether it is better to use existing code outside of the strategy or create an implementation in the strategy is up to the developers to determine, and the choice may come down to gas cost.

Impact

Informational.

Recommendation

Some changes can be made to improve WETH support.

1. To transition to using the Stargate function, change these lines of code: ```diff function _addToLP(uint256 _amount) internal { // Nice! DRY principle _amount = Math.min(balanceOfWant(), _amount); // we don't want to add to LP more than we have // Check if want token is WETH to unwrap from WETH to ETH to wrap to SGETH: if (wantIsWETH == true){
   - _convertWETHtoSGETH(_amount);
   - IWETH(address(want)).withdraw(_amount);
   - stargateRouterETH.addLiquidity{value: _amount}(); } else { // want is not WETH: _checkAllowance(address(stargateRouter), address(want), _amount);
   - stargateRouter.addLiquidity(liquidityPoolID, _amount, address(this)); }
   - stargateRouter.addLiquidity(liquidityPoolID, _amount, address(this)); }

- function _convertWETHtoSGETH(uint256 _amount) internal {
- IWETH(address(want)).withdraw(_amount);
- address SGETH = IPool(address(lpToken)).token();
- ISGETH(SGETH).deposit{value: _amount}();
- _checkAllowance(address(stargateRouter), SGETH, _amount);
- } ```

1. The want token could be verified to match the liquidityPool in the case where `wantIsWETH == true` with this modification in `_initializeThis()`:

```
if (wantIsWETH == false){
    require(address(want) == liquidityPool.token());
+ } else {
+       require(liquidityPoolID == 13); // PoolID == 13 for ETH pool on mainnet,
```

```
Optimism, Arbitrum
}
```

1 Fix this comment which does not match the code because the conversion function is not actually payable.

6. Informational - Documentation improvements possible (engn33r)

The custom `harvestTrigger()` implementation is very similar to the implementation in BaseStrategy. The difference in the custom implementation should be more clearly documented with an explanation of why `super.harvestTrigger()` was not used instead.

Technical Details

The only difference between `harvestTrigger()` in BaseStrategy and Strategy is that the Strategy implementation adds a check for the `minReportDelay` value and uses `> maxReportDelay` instead of `>= maxReportDelay`. Most of the logic in the custom implementation could be replaced by a `super.harvestTrigger()` as suggested in the BaseStrategy NatSpec.

Impact

Informational.

Recommendation

Functions that are mostly borrowed from elsewhere should reference the original code and document the differences to make it easier to understand why the logic is the way it is.

7. Informational - No USDT pool on Optimism (engn33r)

Stargate does not have a USDT Pool.sol contract deployed on Optimism. This strategy is only in use on Ethereum mainnet but may be deployed to Optimism in the future, so this finding notes the inability to port the USDT strategy to Optimism.

Technical Details

A comparison was done of the Stargate contracts on the different chains where Yearn Finance operates (mainnet, Arbitrum, and Optimism). The deployed contracts are listed in the Stargate docs. The findings from this comparison were:

1 Mainnet and Arbitrum contracts of Stargate are identical

2. Optimism has no USDT pool
3. There are two differences between Arbitrum and Optimism a) The USDC Pool.sol on Optimism uses a flattened import structure for some odd reason, but otherwise the logic is the same in the pool contract and all imported files b) The LPStaking contract on Arbitrum and mainnet is renamed to LPStakingTime on Optimism and these contract have many differences: i) The LPStakingTime contract on Optimism uses an "eToken" with emissions for rewards (not STG). The eToken is OP, the Optimism token. It is a standard OpenZeppelin ERC20 token rather than the custom StarGate token that is not a simple OpenZeppelin ERC20. ii) The LPStakingTime contract on Optimism uses `block.timestamp` instead of `block.number` because this is one of the known differences in Optimism. iii) Some functions in LPStakingTime are `external` while the same function in LPStaking is `public`.

Impact

Informational.

Recommendation

The differences in Stargate contracts between different chains should be noted when porting the strategy to other chains. Arbitrum may be easier to port to than Optimism.

8. Informational - `== true` unnecessary for boolean evaluation (engn33r, shung)

There are five places with a boolean is evaluated with `== true` in the contract. A boolean does not need to be compared with true, it can be left as it.

Technical Details

Booleans evaluated with true in the form `require(wantIsWETH == true)` can be simplified to `require(wantIsWETH)`. The longer form may be easier to understand depending on the reader.

Impact

Informational.

Recommendation

An abbreviated approach to evaluating booleans may be used to replace the longform approach.

9. Informational - aeWETH on Arbitrum is not an exact WETH9 clone (engn33r)

aeWETH on Arbitrum is not identical to WETH9 on mainnet.

Technical Details

aeWETH on Arbitrum inherits WETH9 but is not an exact clone of the Ethereum mainnet WETH9 contract like WETH on Optimism. Extra attention may be needed here when deploying the strategy on Arbitrum for WETH.

Impact

Informational.

Recommendation

Consider the differences between tokens and chains when deploying this strategy in new places.

10. Informational - Rename variable (spalen)

Rename variable `_liquidAssets` at L279.

Technical Details

The name `_liquidAssets` is a bit misleading because it represents only a part of liquid assets, i.e. liquidated assets.

Impact

Informational.

Recommendation

Rename to `_liquidatedAssets`.

11. Informational - Use vault decimals (spalen)

At L98 `want` decimals is fetched by importing `IERC20Metadata` but this data is already stored in vault.

Technical Details

Decimals data is stored in the vault. No need to add additional imports.

Impact

Simpler code with less imports.

Recommendation

Replace L98 and remove interface IERC20Metadata import as well as file.

```
    creditThreshold = vault.decimals();
```

12. Informational - Use explicit uint type (spalen)

At L383 and L387 uint alias is used for uint256. The rest of the contract uses explicit uint256.

Technical Details

Style change.

Impact

Informational.

Recommendation

Be consistent and use uint256 for uint.

13. Informational - Unused interface `IPriceFeed` (spalen)

Unused interface `IPriceFeed`.

Technical Details

Interface is defined but not used in the code.

Impact

Informational.

Recommendation

Remove unused file.

14. Informational - Missing protected token (spalen)

In Strategy.sol, function `protectedTokens()` doesn't protect liquidity pool token. By sweeping this token, strategy could end with debt but without any asset.

Technical Details

Sweeping LP token, strategy will report a loss. Only unstaken LP tokens can be sweeped. PoC:

```python
def test_sweep_lp(gov, vault, strategy, token, user, amount, token_lp, chain,
RELATIVE_APPROX):
    # 1- Deposit to the vault
    token.approve(vault.address, amount, {"from": user})
    vault.deposit(amount, {"from": user})
    assert token.balanceOf(vault.address) == amount
    strategy.setDoHealthCheck(False, {"from": gov})

    # 2- Harvest
    chain.sleep(1)
    strategy.harvest({"from": gov})
    assert pytest.approx(strategy.estimatedTotalAssets(), rel=RELATIVE_APPROX) == amount

    # 3- Unstake LP tokens
    strategy.unstakeLP(amount, {"from": gov})

    # 4- Sweep LP tokens
    unprotected_tokens = [token_lp]
    for token in unprotected_tokens:
        strategy.sweep(token, {"from": gov})

    # Strategy lost all of it's money
    assert pytest.approx(strategy.estimatedTotalAssets(), rel=RELATIVE_APPROX) == 0
    # Strategy still has debt
    assert vault.strategies(strategy).dict()["totalDebt"] == amount

    # 5- Report loss
    chain.sleep(1)
    tx = strategy.harvest({"from": gov})
    assert tx.events["StrategyReported"]["loss"] == amount
    assert vault.strategies(strategy).dict()["totalLoss"] == amount
```

Impact

Informational. Impact is minimal because only `governance()` can call function to sweep tokens. Also, tokens will be sent to `governance()` so funds won’t be lost outside yearn ecosystem but strategy would lose its funds without it being reported and left with debt.

Recommendation

Add `lpToken` to the list of protected tokens:

```
    function protectedTokens()
        internal
        view
        override
        returns (address[] memory)
    {
        address[] memory protectedTokens = new address[](1);
        protectedTokens[0] = address(lpToken);
        return protectedTokens;
    }
```

15. Informational - Change function `emergencyUnstakeLP()` visibility (spalen)

The function `emergencyUnstakeLP()` should change visibility to external.

Technical Details

Current visibility is set to public but can be changed to external because strategy uses internal function implementation `_emergencyUnstakeLP()`.

Impact

Informational.

Recommendation

Change function visibility to external.

16. Informational - Reward token can be swept by governance (shung)

Governance could take the reward tokens earned by the strategy.

Technical Details

`BaseStrategy` contract has a virtual function to define which tokens cannot be swept by the governance. The `Strategy` contract does not honor this by not setting the reward token as protected.

Impact

Informational.

Recommendation

Return the `reward` token address in `protectedTokens` function.

17. Informational - Reverts are missing reason string (shung)

Most of the `require` statements in the contract are missing a reason string. Lack of reason string makes it harder to understand the problem when a transaction reverts.

Technical Details

There are five instances of this issue (1, 2, 3, 4, 5).

Impact

Informational.

Recommendation

Add a second argument in `require` statements to return a reason string. Alternatively, use custom errors.

18. Informational - Returned address from `create` is not explicitly checked to be non-zero (shung)

`create` does not revert when the deployment fails, it instead returns zero. It is good practice to explicitly check the return value to be non-zero to ensure deployment did not fail.

Technical Details

In assembly, `create` does not revert when a deployment fails. It instead returns zero address. This is currently not an issue because there is an implicit check due to the call made to the `newStrategy`. However, not having an explicit check might cause this to be overlooked in future refactorings of the code.

Impact

Informational.

Recommendation

Add an explicit check to ensure `newStrategy` address is non-zero. Alternatively, clarify with a comment that the external call to the `newStrategy` also acts a check for ensuring `create` did not fail.

19. Informational - Missing Zero-Address Check (hasanza)

Technical Details

The `lpStaker` assignment in `initializeThis()` is missing a zero address check. Granted, there is an implicit revert when a function is called on zero address; however, debugging an unnamed revert could end up consuming valuable time. As such, a zero address check would expedite Strategy deployment and operation by quickly reverting during the construction of the contract if the LPStaker address passed is a zero address.

Impact

Informational.

Recommendation

Place a zero address check for `_lpStaker` argument inside `initializeThis()`.

20. Informational - Cache storage variables (hasanza)

Technical Details

Accessing a state variable for the first time in a function takes 2100 gas (*Gcoldsload*), and then 100 gas (*Gwarmaccess*) for each additional time. So, it is best to cache (store in a stack/ memory) the storage variable if there are multiple reads for it within the same function.

Impact

Informational.

Recommendation

Cache storage variables that are read more than once in a function. For instance, in `_ldToLp()`, we can make the following change:

```
    function _ldToLp(uint _amountLD) internal returns (uint) {
        // Cache the liquidityPool state variable
        IPool _liquidityPool =  liquidityPool;
        // Use the cached variable in calculations
        require(_liquidityPool.totalLiquidity() > 0);//dev: "Stargate: cant convert
  SDtoLP when totalLiq == 0";
        uint256 _amountSD = _amountLD / _liquidityPool.convertRate();
        return _amountSD * _liquidityPool.totalSupply() /
```

```
_liquidityPool.totalLiquidity();
    }
```

Repeat this change in all functions where there are more than 1 reads of a state variable.

In this way, storage caching can accumulate a lot of gas savings.

21. Informational - Use calldata for unchanging external function args (hasanza)

Technical Details

We can use `calldata` instead of `memory` for `array`, `struct` or `mapping` type arguments in external functions, which are not mutated in the function. This will save gas since the argument would not need to be saved into memory first.

Impact

Informational.

Recommendation

For the `clone` function, change the argument `string memory _strategyName` to `string calldata _strategyName`. This works because `string` is an alias for `bytes1[]` i.e., a `string` is a dynamic array of 1-byte elements. Since `clone` may be called many times as new strategies are cloned from an original one, it will accrue gas savings over time.

22. Informational - Use uint256 for bool values (hasanza)

Technical Details

On the EVM, changing a state variable from `0` to `non-zero` uses *Gsset* and incurs 20,000 gas. In case of bools, every time a bool is changed from `false` (value of 0), to `true`, *Gsset* cost of 20,000 is incurred.

Impact

Informational.

Recommendation

Use `uint256` “bool” variables instead of `bool` types. Then, use `1` for true and `2` for false. For example:

For the following bool variables:

```
    bool public wantIsWETH;
    bool public emissionTokenIsSTG;
    bool internal unstakeLPOnMigration; //if True it would unstake the LP on
`prepareMigration`, if not it would skip this step
```

```
    uint256 public wantIsWETH = 2;
    uint256 public emissionTokenIsSTG = 2;
    uint256 internal unstakeLPOnMigration = 2; //if True it would unstake the LP on
`prepareMigration`, if not it would skip this step
```

23. Informational - Unnecessary declaration for Abi Coder pragma (shung)

Technical Details

Abi Coder v2 is enabled by default since Solidity 0.8.0, hence it is not necessary to explicitly enable it.

Impact

Informational.

Recommendation

Remove the line with encoder pragma.

24. Informational - Constant is not named in capital letters (shung)

Technical Details

The official Solidity style guide recommends using all capital letters for contants. However, the constant `max` does not adhere to this guideline.

Impact

Informational.

Recommendation

Rename the variable accordingly.

25. Informational - Anyone can send ETH to the contract via the `receive()` function (DecorativePineapple)

Technical Details

The `receive()` function allows the Strategy contract to receive ETH from the WETH contract when the want token is WETH. However, anyone can send ETH to the contract via the `receive()` function as the address that sends the ETH isn't checked.

Impact

Informational.

Recommendation

It is recommended to only allow the WETH contract to send ETH to the contract via the receive() function.

```
receive() external payable {
    require(wantIsWETH == true);
+   require(msg.sender == address(want));
}
```

26. Informational - Possible precision loss via the `_ldToLp` function (DecorativePineapple)

Technical Details

In the `_ldToLp` function the value returned is calculated by first dividing and then multiplying. This could lead to precision loss when `liquidityPool.convertRate() != 1`. If the `convertRate()` changes, the `_ldToLp` function will return lower value than expected.

Impact

Informational.

Recommendation

It is recommended to refactor the `_ldToLp` in way that no division will be perform before multiplication.

```
function _ldToLp(uint _amountLD) internal returns (uint) {
    require(liquidityPool.totalLiquidity() > 0);//dev: "Stargate: cant convert SDtoLP
when totalLiq == 0";
-   uint256 _amountSD = _amountLD / liquidityPool.convertRate();
-   return _amountSD * liquidityPool.totalSupply() / liquidityPool.totalLiquidity();
+   return (_amountLD * liquidityPool.totalSupply()) / (liquidityPool.convertRate() *
liquidityPool.totalLiquidity());
}
```

Final remarks

engn33r

The strategy is solid as far as the core USDC and USDT strategy goes. The internal accounting appears sound, as do the strategy's integrations with Stargate and the Yearn Finance vault. It would be best if the low findings were fixed, but they do not appear to impede the strategies normal operations under regular operating conditions. Unfortunately, the Stargate project is not well documented at this time. There is substantial complexity around how `deltaCredit` in the pools work, which is out of scope for this review but an area that the Yearn Finance risk team may wish to evaluate. Some improvements can be made to allow the strategy to work with other chains and tokens but the devs were already aware of this and work is ongoing.

spalen

Knowing that yearn vault code and base strategy are well tested in production, Stargate strategy doesn't expose any additional concerns to outside calls via public functions. All external functions are protected by defined modifiers. The strategy uses Stargate as intended but Stargate protocol documentation is missing a lot of data so deeper analysis can be done only by checking the source code. Stargate as a protocol is not reviewed for potential bugs and problems. The meaning of Delta credit in Stargate is not clear and should be better explained in Stargate docs and strategy tests. Overall, the code is clean but there is room for multiple small gas optimizations and cleaning up unused stuff. Tests are well-written and cover all main cases.

About yAcademy

yAcademy is an ecosystem initiative started by Yearn Finance and its ecosystem partners to bootstrap sustainable and collaborative blockchain security reviews and to nurture aspiring security talent. yAcademy includes a fellowship program, a residents program, and a guest auditor program. In the fellowship program, fellows perform a series of periodic security reviews and presentations during the program. Residents are past fellows who continue to gain experience by performing security reviews of contracts submitted to yAcademy for review (such as this contract). Guest auditors are experts with a track record in the security space who temporarily assist with the review efforts.