CANTINA

Sturdy
Security Review

Cantina Managed review by:
Desmond Ho, Lead Security Researcher
Xmxanuel, Security Researcher

October 19, 2023

Contents

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

Cantina Managed provides a detailed evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While Cantina Managed endeavors to identify and disclose all potential security issues, it cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities that were absent during the initial review. Therefore, any changes made to the code require a new security review to ensure that the code remains secure. Please be advised that the Cantina Managed security review is not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity	Description
Critical	*Directly* exploitable security vulnerabilities that need to be fixed.
High	Security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All high issues should be addressed.
Medium	Objective in nature but are not security vulnerabilities. Should be addressed unless there is a clear reason not to.
Low	Subjective in nature. They are typically suggestions around best practices or readability. Code maintainers should use their own judgment as to whether to address such issues.
Gas Optimization	Suggestions around gas saving practices.
Informational	Suggestions around best practices or readability.

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. When determining the severity one first needs to determine whether the finding is subjective or objective. All subjective findings are considered of Minor severity.

Next it is determined whether the finding can be regarded as a security vulnerability. Some findings might be objective improvements that need to be fixed, but do not impact the project's security overall (Medium).

Finally, objective findings of security vulnerabilities are classified as either critical or major. Critical findings should be directly vulnerable and have a high likelihood of being exploited. Major findings on the other hand may require specific conditions that need to be met before the vulnerability becomes exploitable.

2 Security Review Summary

Sturdy enables anyone to create a liquid money market for any token. Sturdy uses a novel two-tier architecture to isolate risk between assets while avoiding liquidity fragmentation. The base layer consists of risk-isolated pools; aggregation built on top enables lenders to select which collateral assets can be used as collateral for their deposits.

Contracts in scope include:

```
contracts/core/SiloGateway.sol
contracts/core/LenderDebtManager.sol
```

From September 18th to September 21st the Cantina team conducted a review of Sturdyfi on commit hash 6ee0b34f. The team identified a total of **24** issues in the following risk categories:

- Critical Risk: 0
- High Risk: 5
- Medium Risk: 2
- Low Risk: 1
- Gas Optimizations: 5
- Informational: 11

3 Findings

3.1 High Risk

3.1.1 `DebtManager.requestLiquidity` can be exploited to reduce returns of `vault` user

Severity: High Risk

Context: DebtManager.sol#L190

Description: The `SiloGateway.borrowAsset` allows one to request `liquidity` from other `silos` to fulfill a high borrow amount if their own `liquidity` is not sufficient. There are no limitations, on how much liquidity can be requested from other silos. It could be the entire available liquidity across all silos.

This fact can be exploited by an attacker. The attack requires some initial assets but a flash loan could be used as well.

An `attacker` takes a flash loan in the `collateral` of the silo with the lowest `APR`. Afterward, the `SiloGateway.borrowAsset` is called to request all the available liquidity from the other silos. After a successful `borrow` transaction, the open position is immediately repaid.

The repayment will result in a high amount of unused liquidity (`idle`) in the `silo`. The attacker receives the funds back from repayment and pays back the flash loan. In most lending protocols a high amount of `idle` lowers the APR for lenders. In this scenario, the silo with the lowest APR will end up with a large amount of unused liquidity, which will further reduce its APR. All the available liquidity of the aggregator (vault) will be in that silo.

There is also an economic incentive to perform such an attack. If the attacker is a large lender in a `silo` and does not use the vault aggregator, removing the liquidity from the `silo` would result in a higher APR (less idle), thereby increasing the interest earned on the attacker's fund.

Recommendation: Analyze the economic incentive of all potential actors and add potential restrictions to the use of `requestLiquidity`. Another idea could be to charge a `fee` in case a high amount needs to be requested.

Sturdy: Acknowledged. We set some limit of moving fund between silos for JIT liquidity and addressed in ChainSecurity commit 33a4136b.

Cantina: Fixed. There are 2 levels to limit liquidity amounts: a global utilization target limit (denoted as a percentage of total deposits) on the requesting silo, and on the withdrawal amounts from other silos meeting the liquidity demand (capped at each silo's utilization target).

3.1.2 JIT liquidity breaks once first lender has 0 `current_debt`

Severity: High Risk

Context: DebtManager.sol#L244

Description: The internal method `_update_debt()` of Yearn's V3 vault reverts under the following conditions:

- Unchanged debt: `assert new_debt != current_debt, "new debt equals current debt"`
- Reducing debt:
  - `assert withdrawable != 0, "nothing to withdraw"`
  - `assert unrealised_losses_share == 0, "strategy has unrealised losses"`
- Increasing debt:
  - `assert new_debt <= self.strategies[strategy].max_debt, "target debt higher than max debt"`
  - `assert max_deposit != 0, "nothing to deposit"`
  - `assert total_idle > minimum_total_idle, "no funds to deposit"`

The first revert condition is of interest. When the first lender in the `_lenders` array has all its liquidity transferred to other lenders, its `current_debt` becomes 0. Further attempts to transfer liquidity from other lenders will trigger a revert since the function sets `newDebt` of this first lender to 0, effectively equating it to `current_debt`.

Proof of concept: Here's a test case that can be added into `tests/e2e/fraxlend/test_fraxlend_deposit_allocation_requestLiquidity_flow.py`.

```python
import ape
from utils.constants import ROLES

FRAX_CRV_PAIR_ADDRESS = '0x3835a58CA93Cdb5f912519ad366826aC9a752510'
FRAX_CVX_PAIR_ADDRESS = '0xa1D100a5bf6BFd2736837c97248853D989a9ED84'
FRAX_WBTC_PAIR_ADDRESS = '0x32467a5fc2d72d21e8dce990906547a2b012f382'

def test_failing_requestLiquidity_when_first_lender_has_zero_current_debt(frax, crv, gov, accounts,
↪   create_aggregator, create_debt_manager, create_frax_lender, mint, data_provider, create_silo_gateway,
↪   apr_oracle, frax_lend_apr_oracle):
    #############
    ### SETUP ###
    #############
    # deploy crv lender
    crv_lender = create_frax_lender(FRAX_CRV_PAIR_ADDRESS, 'frax/crv lender', '1')
    # deploy cvx lender
    cvx_lender = create_frax_lender(FRAX_CVX_PAIR_ADDRESS, 'frax/cvx lender', '1')
    # deploy wbtc lender
    wbtc_lender = create_frax_lender(FRAX_WBTC_PAIR_ADDRESS, 'frax/wbtc lender', '1')

    # deploy FRAX aggregator
    silo_datas = [
        {
            "lender": crv_lender.address,
            "maxDebt": 50_000 * 10 ** 18,
        },
        {
            "lender": cvx_lender.address,
            "maxDebt": 50_000 * 10 ** 18,
        },
        {
            "lender": wbtc_lender.address,
            "maxDebt": 50_000 * 10 ** 18,
        }
    ]
    aggregator, accountant = create_aggregator(frax, silos=silo_datas)
    # deploy debt manager
    manager = create_debt_manager(aggregator)
    # set oracle
    apr_oracle.setOracle(crv_lender.address, frax_lend_apr_oracle.address, sender=gov)
    apr_oracle.setOracle(cvx_lender.address, frax_lend_apr_oracle.address, sender=gov)
    apr_oracle.setOracle(wbtc_lender.address, frax_lend_apr_oracle.address, sender=gov)
    aggregator.set_role(
        manager.address,
        ROLES.DEBT_MANAGER
        | ROLES.REPORTING_MANAGER,
        sender=gov,
    )
    # add CRV, CVX FraxLender to manager
    manager.addLender(crv_lender.address, sender=gov)
    manager.addLender(cvx_lender.address, sender=gov)
    manager.addLender(wbtc_lender.address, sender=gov)
    # deploy silo gateway
    silo_gateway = create_silo_gateway(manager)
    manager.setWhitelistedGateway(silo_gateway.address, True, sender=gov)
    manager.setPairToLender(FRAX_CRV_PAIR_ADDRESS, crv_lender.address, sender=gov)
    manager.setPairToLender(FRAX_CVX_PAIR_ADDRESS, cvx_lender.address, sender=gov)
    manager.setPairToLender(FRAX_WBTC_PAIR_ADDRESS, wbtc_lender.address, sender=gov)

    # === User Deposits 30000 FRAX
    user1 = accounts[1]
    depositAmount = 30000 * 10 ** 18
    # prepare FRAX
    mint('FRAX', depositAmount, user1)
    # approve aggregator
    frax.approve(aggregator.address, depositAmount, sender=user1)
    # deposit
    aggregator.deposit(depositAmount, user1.address, sender=user1)

    # Allocations
    positions = [
      {
        "lender": crv_lender.address,
        "debt": 10000 * 10 ** 18
      },
      {
        "lender": cvx_lender.address,
        "debt": 10000 * 10 ** 18
      },
      {
        "lender": wbtc_lender.address,
        "debt": 10000 * 10 ** 18
      }
    ]
    manager.manualAllocation(positions, sender=gov)

    # === Sort lenders: [WBTC_LENDER, CVX_LENDER, CRV_LENDER]
    manager.sortLendersWithAPR(sender=gov)
    lenders = manager.getLenders()
    assert lenders[0] == wbtc_lender.address
    assert lenders[1] == cvx_lender.address
    assert lenders[2] == crv_lender.address

    # Change utilizationLimit to 74.2% to make requestLiquidity call happen
    silo_gateway.setUtilizationLimit(74_200, sender=gov)

    # === user5 through gateway contract borrow 5000 FRAX from crv_lender and success
    # should pull all liquidity from wbtc_lender, leaving it with 0 new_debt
    user5 = accounts[5]
    borrowAmount = 5000 * 10 ** 18
    # Mint CRV for collateral
    mint('CRV', borrowAmount * 8, user5)
    # Approve for collateral
    crv.approve(silo_gateway.address, int(borrowAmount * 8), sender=user5)
    # 20K collateral deposit and Borrow 5K
    silo_gateway.borrowAsset(FRAX_CRV_PAIR_ADDRESS, borrowAmount, int(borrowAmount * 8), crv.address,
    ↪  user5.address, sender=user5)
    # At this point however, it has drained the first lender
    assert aggregator.strategies(wbtc_lender.address).current_debt == 0

    # attempt to borrow again, will fail because first lender has already been drained
    borrowAmount = 1000 * 10 ** 18
    with ape.reverts("new debt equals current debt"):
       silo_gateway.borrowAsset(FRAX_CRV_PAIR_ADDRESS, borrowAmount, 0, crv.address, user5.address,
       ↪  sender=user5)
```

Recommendation: `continue` if `current_debt` is 0:

```
- if (lenders[i] == requestingLender) continue;
+ if (lenders[i] == requestingLender || lenderData.current_debt == 0) continue;
```

Sturdy: Addressed in commit 3109aad2.

Cantina: Fixed. It is noted that these checks have been refactored and shifted to `_getAvailableAmountsAndDatas()` in subsequent commits.

3.1.3 Prevent griefing attack on `DebtManager.manualAllocation`

Severity: High Risk

Context: DebtManager.sol#L305

Description: The manual allocation will be always calculated based on a system state `s`. After the transaction gets minted there could be a completely new state `s'`.

An attacker could try to frontrun the `manualAllocation` transaction to produce a revert. There may be a financial incentive for the attacker to do so.

The manual allocation aims to optimize resource allocation by keeping only the `minimum_total_idle amount`, along with an optional buffer, in `idle`. An attacker could exploit this by withdrawing an amount equal to (buffer + 1) from the vault, thereby triggering a revert in the `manualAllocation`.

The revert in the manual allocation would happen in the `vaultV3.update_debt` function (see VaultV3.vy#L992).

A silo position `debt_update` would use all the remaining funds from the `idle`. The following silo `debt_-update` position in the loop would revert because `total_idle == minimum_total_idle`.

Recommendation: Add the following check to the `manualAllocation` loop.

```
// deposit/increase not possible because minimum total idle reached
if (position.debt > lenderData.current_debt &&
aggregator.totalIdle() == aggregator.minimum_total_idle()) continue;
```

Sturdy: Addressed in commit 9e1fa3cb.

Cantina: Fixed, although the recommendation should be have been as follows:

```
- // deposit/increase not possible because minimum total idle reached
+ // deposit/increase not possible because minimum total idle not reached
  if (position.debt > lenderData.current_debt &&
- aggregator.totalIdle() == aggregator.minimum_total_idle()) continue;
+ aggregator.totalIdle() <= aggregator.minimum_total_idle()) continue;
```

3.1.4 Collateral & debt position may be accounted to `SiloGateway` instead of user

Severity: High Risk

Context: SiloGateway.sol#L92-L96

Description: `borrowAsset()` intends to add collateral and borrow the asset on behalf of the user, but this feature may not be supported by some protocols. For instance, some protocols will accrue the collateral and debt to the caller, which would be `SiloGateway`.

An instance would be `FraxLend`, which was one of the protocols used for e2e tests. The code snippet below is taken from the `borrowAsset()` of `FraxLendPair`.

```
if (_collateralAmount > 0) {
  _addCollateral(msg.sender, _collateralAmount, msg.sender);
}

function _addCollateral(
  address _sender,
  uint256 _collateralAmount,
  address _borrower
) internal {
userCollateralBalance[_borrower] += _collateralAmount;
```

This results in the user's collateral being permanently locked up, even if debt is repaid on behalf of the contract.

Recommendation: Protocols need to be carefully checked to ensure that borrowing on behalf of the user is supported.

Sturdy: Acknowledged (see commit b0a71073. We will use this contract when we deploy our `sturdy v2` silos which has the feature of borrowing on behalf of the user, or implement another gateway contract for the `Aave V3` and `Compound V3` silos.

Cantina: Acknowledged.

3.1.5 `DebtManager.requestLiquidity` from other silos even if not needed at all

Severity: High Risk

Context: DebtManager.sol#L218

Description: The `requestLiquidity` function allows to request liquidity from other silos to fulfill a `borrowAsset` operation by the user. A silo could be any other lending protocol that implements the `ERC4626` standard.

The aggregator, a `YearnVaultV3`, is responsible for managing these silos. It maintains a balance of uninvested funds, termed `totalIdle`, which does not earn interest.

```
if (requiredAmount > totalIdle) {
    unchecked {
    requiredAmount -= totalIdle;
    }
}
```

First, the `totalIdle` should be used to fulfill the liquidity request. If more liquidity is needed it should be requested from the `silos`. After the `if` statement the loop iterates over the silos to fulfill the remaining `requiredAmount`.

However, in case the `requiredAmount <= totalIdle`, the `requiredAmount` could be fulfilled directly from the `totalIdle`. No additional silo requests are needed.

Otherwise, too many funds are withdrawn from the silos and will stay in `totalIdle` resulting in a loss of interest.

This case is currently not considered. The full `requiredAmount` will be requested from silos in case `requiredAmount <= totalIdle`.

Recommendation: Only request liquidity from silos if `requiredAmount > totalIdle`. The loop iteration over the silos should happen inside the `if` statement:

```
if (requiredAmount > totalIdle) {
    unchecked {
        requiredAmount -= totalIdle;
    }
    for (uint256 i; i < lenderCount; ++i) {
    // ...
    }
}
```

Sturdy: Addressed in ChainSecurity commit e1c04047.

Cantina: Fixed.

3.2 Medium Risk

3.2.1 `DebtManager.manualAllocation` is not considering the respect minimum idle feature in `VaultV3.update_debt`

Severity: Medium Risk

Context: DebtManager.sol#L300

Description: The `manualAllocation` allows to newly allocate the `debt` for each silo in the `VaultV3` by increasing or decreasing it.

The `VaultV3.update_debt` has a feature to respect the minimum total idle (see VaultV3.vy#L935). This constraint defines the minimum amount of assets in the `VaultV3` which should not be invested into silos.

This means in the `decrease/withdraw` debt from silo case, if the minimum total idle constraint is violated it would withdraw more funds than acutally requested from the `update_debt` call.

This fact is not considered in the `manualAllocation` function. It assumes the passed param `new debt` to the `update_debt` will change the debt accordingly, which can lead to a revert of the `manualAllocation` function.

Recommendation: The array passed into the `manualAllocation` function needs to consider the minimum idle constraint for each position. An alternative solution could be to consider the actual new debt which is returned by the `update_debt` function. A difference here needs to be considered in the upcoming `update_-debt` calls for the `successor`.

Sturdy: Acknowledged. When `zkVerifier` or `admin` makes the allocation data off-chain, it requires to consider the `minimum_idle_amount` and every silo's `maxDebt` amount to avoid revert case and allocate exactly.

Cantina: Acknowledged.

3.2.2 Newly added `lender` without an `aprOracle` entry would block `DebtManager.sortLendersWithAPR`

Severity: Medium Risk

Context: DebtManager.sol#L119

Description: Each `lender` added to the `DebtManager` requires an entry in the `AprOracle` contract. If a newly added lender doesn't have an oracle entry in the `AprOracle` it would block the entire sorting because `getExpectedApr` would revert.

For context, see the comment added within the `AprOracle` contract below:

```
contract AprOracle {
    mapping(address => address) public oracles;

    function getExpectedApr(
        address _strategy,
        int256 _debtChange
    ) external view returns (uint256) {
        address oracle = oracles[_strategy];

        // Will revert if a oracle is not set.
        return IOracle(oracle).aprAfterDebtChange(_strategy, _debtChange);
    }

    function setOracle(address _strategy, address _oracle) external {
        require(msg.sender == IStrategy(_strategy).management(), "!authorized");

        oracles[_strategy] = _oracle;
    }
}
```

Recommendation: Consider enforcing the `aprOracle` has an entry before adding the `lender` to the `Debt-Manager`.

Sturdy: Addressed in commit a1357196.

Cantina: Fixed.

3.3 Low Risk

3.3.1 Enforce correct role management for `DebtManager`

Severity: Low Risk

Context: DebtManager.sol#L241

Description: The `DebtManager` needs to hold multiple roles in the `VaultV3` to be able to call the needed `Vault` functions:

- `Roles.REPORTING_MANAGER` for `process_report`
- `Roles.DEBT_MANAGER` for `update_debt`

There is currently no factory contract nor another function that ensures the correct setup of the `DebtMan-ager`.

It is also not clear which address is assigned to the `role_manager` in the `VaultV3`. This is only the `msg.sender` in the `AggregatorFactory`.

Recommendation: Create a `FactoryContract` which ensures the correct setup of a new `VaultV3` with the `DebtManager`.

Sturdy: Acknowledged. `role_manager` should be owner of the `Aggregator` (the vault). Everyone can create their own aggregator via `AggregatorFactory` and in this case the creator would be `role_manager`. Regarding the debt manager, it can be the specific contract or user itself. In other words, the creators can make their own debt manager contract or manually manage the debt of aggregator via calling `update_debt()` manually.

Cantina: Acknowledged.

3.4 Gas Optimization

3.4.1 `selection sort` with stop condition instead of `bubble sort` wouldn't require to sort the entire list in `sortLendersWithAPR`

Severity: Gas Optimization

Context: DebtManager.sol#L119

Description: One alternative design instead of bubble sort could be to directly sort in `requestLiqudity` with `selection sort`.

This would be more efficient because it wouldn't require sorting the entire list. Selection sort could track the amount of the already sorted list elements and stop after requiredAmount is reached.

Recommendation: Consider `selection sort` in the `requestLiquitity` function and track the amount available in sorted silos and stop after the first n elements if the required amount is reached. The choice of the right sorting algorithm depends on the number of expected elements in the lender array. If only a low number is expected, a simple bubble sort might be sufficient.

Sturdy: Addressed in commit 6a2b1461. Since we use selection sort, removed the `sortWithLenderAPR` function.

Cantina: Fixed. Selection sort is now used to search for and select the most appropriate silo.

3.4.2 Remove oracle calls from the inner loop in `DebtManager.sortLendersWithAPR`

Severity: Gas Optimization

Context: DebtManager.sol#L128

Description: It could be cheaper from a gas perspective collecting first the `apr` and storing them in memory instead of performing multiple contract calls to the oracle within the inner loop to request them.

```
oracle.getExpectedApr(lenders[i], 0)
```

Recommendation: First, iterate once over the lender to calculate the `apr` and store them in memory. Afterwards, sort based on the `apr`.

Sturdy: Acknowledged. We will change the sort logic based on the silo's available withdrawal amount, not the `apr` value. In this case, we will follow the recommendation to reduce gas.

Cantina: Acknowledged.

3.4.3 Variable swapping can be more efficient

Severity: Gas Optimization

Context: DebtManager.sol#L131-L133

Description: Solidity has a way to swap 2 variable values in a single line that's more gas efficient, instead of requiring a 3rd variable.

Recommendation: Implement Solidity's native variable swapping:

```
- address temp = lenders[i];
- lenders[i] = lenders[j];
- lenders[j] = temp;

+ (lenders[i], lenders[j]) = (lenders[j], lenders[i]);
```

Sturdy: Acknowledged. This part would be removed since we are going to implement the selection sort with the stop condition.

Cantina: Fixed in the implementation of selection sort.

3.4.4 Unnecessary calculation when `lenderData.current_debt == requiredAmount`

Severity: Gas Optimization

Context: DebtManager.sol#L234-L238

Description: If `lenderData.current_debt == requiredAmount`, the calculated `newDebt` will be 0, which is the initial value.

Recommendation: Drop the equality case.

```
- if (lenderData.current_debt >= requiredAmount) {
+ if (lenderData.current_debt > requiredAmount) {
```

Sturdy: Addressed in commit 35e304db.

Cantina: Fixed.

3.4.5 `_manualAllocation()` can be unchecked

Severity: Gas Optimization

Context: DebtManager.sol#L300-L332

Description: The mathematical operations performed in the `_manualAllocation()` function are:

- `++i` for-loop increment
- `lenderData.current_debt - position.debt` which has been safety checked in the line above it.

Recommendation: The entire function can be wrapped in an unchecked block.

Sturdy: Addressed in commit 8ba7fe4e.

Cantina: Fixed.

3.5 Informational

3.5.1 Smaller improvements and cleanup recommendations

Severity: Informational

Context: DebtManager.sol

Description: This issue outlines a couple of small improvements that may be applied to the codebase.

Recommendation: Consider applying the following recommendations:

- **Unused Imports**:

```
import {IERC4626} from "@openzeppelin/contracts/interfaces/IERC4626.sol";
```

Consider removing this import.

- **Public State variables instead of getter functions**: using a public state variable instead of `getFunction` is a common pattern in Solidity. This can also be applied to the `_whitelistedGateway` and `_pairToLender` mappings.
- **Higher Solidity version**: consider upgrading to a more recent Solidity version than `0.8.18`
- **More granular Error Types**: for example `AG_NOT_ZK_VERIFIER` if a caller is not the zkVerifier.

Sturdy: Addressed in commit 18d5a4ae.

Cantina: Fixed.

3.5.2 Consider a `manualAllocation` role for `DebtManager`

Severity: Informational

Context: DebtManager.sol#L157

Description: In the current implementation the `DebtManager` has three roles:

- `owner`
- whitelistedGateway `mapping`
- `zkVerifier`

Only the `owner` can call `DebtManager.manualAllocation`

Recommendation: Adding and removing new `lenders` is a very powerful operation and should require multiple signatures or a governance process. On the other hand the ability to call `manualAllocation` is still a trusted operation but it has no indirect access to funds. Therefore, consider adding another role called `manualAllocation` role.

Sturdy: Addressed in commit 094f6b9a.

Cantina: Fixed. A manual allocator was added.

3.5.3 Assumption of direction correlation between lending APR and utilisation may be false

Severity: Informational

Context: SiloGateway.sol#L76-L81

Description: The JIT liquidity feature allows shifting of liquidity from other lenders to the requested lender should the utilisation rate exceed the utilisation target, so that the lent liquidity benefits from higher lending APR. In other words, there is an assumption made that greater utilisation rates is directly correlated with lending / borrow APRs.

While this correlation may be true for AaveV3 and CompoundV3 kink interest rate models, it is not the case for a time weighted interest rate / variable rate v2 model that FraxLend uses.

Looking at the pair dashboard, at the time of writing, `gOHM/FRAX` has 67.53% utilisation with 8.8% lending APR while `sfrxETH/FRAX` has 74.11% utilisation with 2.1% lending APR.

Hence, there could be sub-optimal liquidity allocation as a result of the JIT feature.

Recommendation: The direct correlation assumption between lending APR and utilisation should be verified when integrating a new silo.

Sturdy: Acknowledged.

Cantina: Acknowledged.

3.5.4 Variable naming inconsistencies

Severity: Informational

Context: DebtManager.sol#L11

Description: Currently, there are multiple different names used to refer to the same thing in the codebase.

A `silo` is called `strategy` in the vault and is sometimes referred to as `pool` in the contracts. The `Vault` is called `Aggregator` in DebtManager.

Recommendation: Use the names defined by Yearn Finance in the Vault.

Sturdy: Addressed in commit 77c364a4.

Cantina: Fixed.

3.5.5 Allowing more `maximum` and `limit` values to make the `DebtManager` more robust

Severity: Informational

Context: DebtManager.sol#L11

Description: Adding more constraints and limits to the `DebtManager` would make the system more stable.

Recommendation: Consider adding the following constraints to the system:

- `maxAPR` **for silos**: if a silo has an `APR` that exceeds the `maxAPR`, cease all interactions with it. No additional `liquidityRequests` or allocations should be made. For instance, an `APR` of one million percent would suggest that something is fundamentally flawed within the silo or that the interest rates have been incorrectly set.
- `silo.upperLimit` **for** `requestLiqudity`: there could be a `upperLimit` on how much a silo is allowed to request depending on its `utilisiationRate`. Currently, there only exists a `max_debt`
- `upperLimit` **for all silos for** `requestLiqudity`: there could be a `upperLimit` in percentage of `vault.totalAssets` for `requestLiquidity`. Currently, all the available `liquidity` could be used to fulfill a `requestLiqudity`.
- `availableForRequests` **per silo**: a boolean flag that can be set for each silo by the `owner`, if it is available for `requestLiquidity` from other silos. The default could be `true`.
- `SiloGatway.setUtilizationLimit` **max value**: if a new utilization limit is set. Check if it is lower or equal to `ONE`.

Sturdy: We added some more constraints and addressed in ChainSecurity commit 33a4136b. It has the `utilizationLimit` per silo, total utilization limit of vault (aggregator).

Cantina: Fixed. Silo-specific utilization limits and a global utilization limit was added.

3.5.6 Increase granularity of utilization target

Severity: Informational

Context: SiloGateway.sol#L23

Description: The current implementation sets `_utilizationLimit` at a global `SiloGateway` level. However, markets with the same `asset` but different collaterals are likely to have different risk profiles and parameters, so it's a viable use-case to enable setting silo-specific utilization limits (targets).

Recommendation: Consider adding silo-specific utilization targets that defaults to the global utilization target.

Sturdy: Addressed in ChainSecurity commit 33a4136b.

Cantina: Fixed.

3.5.7 Rename `_utilizationLimit` to `_utilizationTarget`

Severity: Informational

Context: SiloGateway.sol#L23

Description: Based on its purpose and usage in `borrowAsset()`, a better variable name for `_utilizationLimit` would be `_utilizationTarget`: if the expected utilization rate exceeds the target, liquidity is pulled from other lenders to bring it down to that level.

Recommendation: Replace `_utilizationLimit` with `_utilizationTarget`.

Sturdy: Addressed in commit f649a6df.

Cantina: Fixed.

3.5.8 Redundant `asset` variable in `DebtManager`

Severity: Informational

Context: DebtManager.sol#L13

Description: `asset` is set as an immutable in the `DebtManager`, but it is neither part of an interface override, nor used anywhere within the contract.

Recommendation: Consider removing the variable, or shift asset validation in `SiloGateway` to this contract to utilise it.

Sturdy: Addressed in commit fbbe2ebf.

Cantina: Fixed. The variable has been removed.

3.5.9 Spelling Error

Severity: Informational

Context: DebtManager.sol#L165

Description: A spelling mistake has been caught.

Recommendation: Install the Spell Code Checker VSCode extension.

```diff
- verifer
+ verifier
```

Sturdy: Addressed in commit f2c952b7.

Cantina: Fixed.

3.5.10 `position.lender` isn't verified to be within `_lenders` array

Severity: Informational

Context: DebtManager.sol#L300-L332

Description: `_manualAllocation()` checks the validity of a lender (strategy), but not if it already exists in the `_lenders` array. Hence, there could be a scenario where the lender has not been added, or has been removed, from the array for other strategists to manage, but avoid being managed via the `_zkVerifier`.

However, since the check doesn't exist, it would still be possible to manage such strategies through the `_zkVerifier`. For instance, consider 2 lenders A and B, where `_lenders = [lenderB]`. Anyone could modify `lenderA`'s allocation still, with `[lenderA]` as the input.

Recommendation: Consider verifying that the lender has been included in the `_lenders` array.

Sturdy: Addressed in commit 362bed73.

Cantina: Fixed.

3.5.11 General lack of event emission in `DebtManager` and `SiloGateway`

Severity: Informational

Context: DebtManager.sol#L107

Description: There is a general lack of event emission in `DebtManager` and `SiloGateway`.

Recommendation: For the sake of transparency and on-chain monitoring, consider using events for major operations like `setAprOracle`, `setWhitelistedGateway`, `setZKVerifier`, etc...

Sturdy: Addressed in commit 9424c323.

Cantina: Fixed.