Sommelier A-1

Security Audit

July 7, 2022

Version 1.0.0

Prepared by

0xMacro.com

Introduction

This document includes the results of the security audit for Sommelier's smart contract code as found in the section titled 'Source Code'. The initial security audit was performed by the Macro Audits team from May 9, 2022 to May 23, 2022. Initial findings caused Sommelier to make significant changes and rewrite one of the contracts. As a result, both parties agreed upon a fresh restart to the audit, which proceeded from June 6, 2022 to June 27, 2022. Findings within this document are with respect to the period beginning June 6.

The purpose of this audit is to review the source code of certain Sommelier Solidity contracts, and provide feedback on the design, architecture, and quality of the source code with an emphasis on validating the correctness and security of the software in its entirety.

Disclaimer: While Macro's review is comprehensive and has surfaced some changes that should be made to the source code, this audit should not solely be relied upon for security, as no single audit is guaranteed to catch all possible bugs.

Overall Assessment

Number of issues reported by type:

Severity	Count	Fixed	Won't Fix	Acknowledged
High	1	1		
Medium	3	2	1	
Low	2	2		
Informational	4			4
Code Quality	1	1		
Gas Optimization	8	6		2

Sommelier was quick to respond and fix these issues.

Specification

Our understanding of the specification was based on the following sources:

- Discussions on Telegram with the Sommelier team.
- The official website and docs.

Source Code

The following source code was reviewed during the audit:

Repository	Commit
Github (cellar-contracts)	4409661be89308e5c3a60f84e898bc068337b13d

Specifically, we audited the following contracts within the **cellar-contracts** repository:

Contract	Sha256
src/AaveV2StablecoinCellar.sol	055be9f450c0c949592dfe2a0fb7e6c7d8e0f177245f3c53e4e83b67cd71a47e
src/CellarStaking.sol	100f13fd3fc634cdae7eace2dad8e3a58bda9578e917577532ce70b21e55e6c6
src/Errors.sol	5d239b63ba094107f14b987b767ae53ac48aae12fab8b97961bd982386605619
src/base/ERC4626.sol	8575cd6db9ec73f5edc75405da6c07b7996d08f1dc17d3e5d6f3e8fc4306cb06
src/base/Multicall.sol	a7ea603501f98e4bf67287480025c7b2b091487b76b3da096651a2709cb3f245
src/interfaces/IAToken.sol	8252f9a8cc6c45486e94dcacba8a004ccb0e48b25e0688b422f0a3f6d6c0945c
src/interfaces/IAaveIncentivesController.sol	9ee7fe06a151148bdcadbe4d1974f0696219e20995b90d69e5a8acf86f2a4cdf

src/interfaces/IAaveProtocolDataProvider.sol	**cc6c4bb9a2f88469f139f1d0653094d00f455afdd59591a5c494fe14957d4ee8**
src/interfaces/IAaveV2StablecoinCellar.sol	**f783819df00bc155c3cd7e67b3c5dc8206a8756dbb0ec5a8271f715bd0f31f54**
src/interfaces/ICellarStaking.sol	**3a551e9e614103f71c0d28d61acff876d8a4fb2c29c54218eee085a27c73ccfd**
src/interfaces/ICurveSwaps.sol	**1da52dd8cd91dd9e0a5cf932b621d01af216a4a4f9a82038a32774f808214c91**
src/interfaces/IERC20.sol	**17a4bbdf77e4c2f67dfb2d17a949e84bf4674f2e034555092e981e75fedce7dc**
src/interfaces/IERC4626.sol	**82c0da66395326fff1d5c52604ec9e6ab5849fc220d3fdb8a4e3bd8b866404f3**
src/interfaces/IGravity.sol	**d0b85433f383d57d88f5e9ca5b79b3d4f012f8562b6b43f2a452eaae7b33220c**
src/interfaces/ILendingPool.sol	**88659c5a9954c018dc15aca21a9554ba1d6840332eff127761e5b1fe57839d11**
src/interfaces/IMulticall.sol	**0d976a972ff63255c5b191b610d7aabe2c43e2918db8e37ee15a28ed3f958586**
src/interfaces/IStakedTokenV2.sol	**93b5f4a46726c4710bf2dfbf8ed4a0026ea41618d31951cc7f956b9aa534b860**
src/interfaces/ISushiSwapRouter.sol	**1ba08b8ef06fd3d31e6d065d5c7c3f33ea43bacf117e7106b5eb92e1ec9ee95e**
src/utils/Math.sol	**2526294e38c181a08e614dc2edac3deecc6b7e4fae9cfe2e6aed360256d55f62**

Note: This document contains an audit solely of the Solidity contracts listed above. Specifically, the audit pertains only to the contracts themselves, and does not pertain to any other programs or scripts, including deployment scripts.

Methodology

The audit was conducted in several steps.

First, we reviewed in detail all available documentation and specifications for the project, as described in the 'Specification' section above.

Second, we performed a thorough manual review of the code, checking that the code matched up with the specification, as well as the spirit of the contract (i.e. the intended behavior). During this manual review portion of the audit we primarily searched for security vulnerabilities, unwanted behavior vulnerabilities, and problems with systems of incentives.

Third, we performed the automated portion of the review consisting of assessing the quality of the test suite and evaluating the results of various symbolic execution tools against the code.

Lastly, we performed a final line-by-line inspection of the code – including comments –in an effort to find any minor issues with code quality, documentation, or best practices.

Issues Descriptions and Recommendations

Severity Level Reference

Level	Description
High	The issue poses existential risk to the project, and the issue identified could lead to massive financial or reputational repercussions. We highly recommend fixing the reported issue. If you have already deployed, you should upgrade or redeploy your contracts.
Medium	The potential risk is large, but there is some ambiguity surrounding whether or not the issue would practically manifest. We recommend considering a fix for the reported issue.
Low / Informational	The risk is small, unlikely, or not relevant to the project in a meaningful way. Whether or not the project wants to develop a fix is up to the goals and needs of the project.
Code Quality	The issue identified does not pose any obvious risk, but fixing it would improve overall code quality, conform to recommended best practices, and perhaps lead to fewer development issues in the future.
Gas Optimizations	The presented optimization suggestion would save an amount of gas significant enough, in our opinion, to be worth the development cost of implementing it.

[H-01] Stakers from previous reward cycles are awarded more rewards than expected when a new reward cycle starts

~~HIGH~~ *Fixed by 3e5be2591201e6d8008ce9f64ccd2cd6cce78715*

In CellarStaking.sol, the owner starts a new reward cycle by calling `notifyRewardAmount()`. The intention of the owner is to distribute the reward to the stakers over a time period with a known duration. Given a reward amount and a time period the accounting model calculates a reward per token per second and updates such reward per token at every accounting checkpoint.

However, the code doesn't consider the new accounting checkpoint when a new cycle begins, allowing stakers of the previous cycles to take advantage of the mis-calculation of `rewardPerTokenStored` and claim rewards that they haven't earned.

By not updating the accounting checkpoint, the `rewardPerTokenStored` calculation accounts for idle times between rewards cycles when it shouldn't. This can lead to a complete drain of tokens from the contract balance if enough time has passed between cycles.

Consider updating the value of `lastAccountingTimestamp = block.timestamp` whenever `notifyRewardAmount()` is called.

[M-01] Inaccurate platform fee accounting

MEDIUM *Won't fix (see response from Sommelier below for details)*

In AaveV2StablecoinCellar.sol, `platformFeeInAssets` is calculated in part by `balanceThisAccrual * elapsedTime`, where `elapsedTime` is the time since the last call to `accrue()`. This opens the door for platform fee accounting to yield inaccurate fees. See the `accrue()` function, lines 498-504:

```
// Get balance since last accrual and updated balance for this accrual.
uint256 balanceLastAccrual = totalBalance;
uint256 balanceThisAccrual = assetAToken.balanceOf(address(this));

// Calculate platform fees accrued.
uint256 elapsedTime = block.timestamp - lastAccrual;
uint256 platformFeeInAssets = (**balanceThisAccrual * elapsedTime** *
platformFee) / 1e18 / 365 days;
```

For additional information see the corresponding "[M-01]" Appendix entry.

Consider updating accounting logic to more precisely consider durations of changes to `totalBalance`, and incorporate this into platform fees calculation.

Response from Sommelier:
"Although a more fine-grained accounting is obviously preferable, the alternative of needing to do platform fees accounting and storing/updating in state everytime TVL updates makes any function that changes TVL (which is basically all of them) more expensive. Additionally, upon doing research to see how other protocols (eg. Yearn, Enzyme, Set) take platform fees we found they all do it a similar way to how we've done here so we are comfortable keeping it as is even though we acknowledge we are sacrificing granularity and accuracy of accounting of the platform fees minted."

[M-02] Unanchored performance fee accounting

~~MEDIUM~~ *Fixed by 3e5be2591201e6d8008ce9f64ccd2cd6cce78715*

The AaveV2StablecoinCellar.sol performance fee accounting within `accrue()` lacks fairness controls in loss-accounting scenarios. It is - for example - neither anchored to a high water mark, nor restricted to a definite measurement period. As a result, performance fee collection can be accumulated in excess of actual performance gains; within a net loss; and be taken repeatedly for a single trading range at any timescale.

The `accrue()` function, lines 506-509:

```
// Calculate performance fees accrued.
uint256 yield = **balanceThisAccrual.subMinZero(balanceLastAccrual)**;
uint256 performanceFeeInAssets = yield.mulWadDown(performanceFee);
uint256 performanceFees =
performanceFeeInAssets.mulDivDown(exchangeRate, oneAsset); // Convert
to shares.
```

`yield` is calculated purely by the difference between `balanceThisAccrual` and `balanceLastAccrual`. This means that any time the prior balance was less than current balance, a performance fee is taken.

Consider the following scenario:

- Block A
  - `accrue()` is called: `totalBalance` is $1000
- Block M (1 day after Block A)
  - A $100 loss in aUSDC yield occurs
  - `accrue()` is called: negative performance, and new `totalBalance` is $900
- Block X (1 day after Block B)
  - A $100 gain in aUSDC yield occurs
  - `accrue()` is called: positive performance with yield of $10, and new `totalBalance` is $1000

In the 2-day span there is a net $0 performance gain, yet $10 in yield was extracted as a performance fee. This pattern can occur at any time scale - the choice of 2 days is arbitrary. The above characteristics can be repeated for every loss-gain cycle, allowing performance fees to be re-extracted repeatedly within the same range, whether a net gain has occurred or not. This occurs because there is no fairness control mechanism in the performance fee calculation: performance gain is seen solely as the balance increase between calls to `accrue()`.

Note: this issue is only a problem in the event of AAVE not being lossless.

Consider adding proper fairness controls, such as high water mark accounting or instituting a definite measurement period. Consider doing so with fairness of the individual investor in mind given various market cycles and conditions.

[M-03] Unexpected asset gain/loss with fee-on-transfer assets

~~MEDIUM~~ *Fixed by 3e5be2591201e6d8008ce9f64ccd2cd6cce78715*

AaveV2StablecoinCellar.sol does not validate transfers from the asset ERC20. As a result, a fee-on-transfer asset can cause asset gains/losses in an unexpected manner.

The AaveV2StablecoinCellar is a tokenized vault (ERC4626) with the ability to dynamically support arbitrary ERC20 assets. While the ERC4626 implementation grants vault shares as a result of transfers from an external ERC20, it does so without checking what balance was actually transferred. Should the asset implement fee-on-transfer, the actual assets transferred in will not correspond to the shares returned. See the base/ERC4626.sol implementation of `deposit()` and `mint()`, both of which transfer assets and then mint vault shares, lines 47-76:

```
function deposit(uint256 assets, address receiver) public virtual
returns (uint256 shares) {
        // Check for rounding error since we round down in
previewDeposit.
    require((shares = previewDeposit(assets)) != 0, "ZERO_SHARES");

    beforeDeposit(assets, shares, receiver);

    // Need to transfer before minting or ERC777s could reenter.
    asset.safeTransferFrom(msg.sender, address(this), assets);

    _mint(receiver, shares);

    emit Deposit(msg.sender, receiver, assets, shares);
```

```
        afterDeposit(assets, shares, receiver);
    }

    function mint(uint256 shares, address receiver) public virtual returns
    (uint256 assets) {
        assets = previewMint(shares); // No need to check for rounding
    error, previewMint rounds up.

        beforeDeposit(assets, shares, receiver);

        // Need to transfer before minting or ERC777s could reenter.
        asset.safeTransferFrom(msg.sender, address(this), assets);

        _mint(receiver, shares);

        emit Deposit(msg.sender, receiver, assets, shares);

        afterDeposit(assets, shares, receiver);
    }
```

If fee-on-transfer logic applies a different fee percentage to different users, or if the Cellar rebalances among assets with/without fee-on-transfer characteristics, assets can be gained/lost by inaccurate provisioning of shares.

Consider the former case, where two users deposit CoinA which applies a variable 0-2% fee on transfer. Assume a 1:1 share/asset conversion ratio.

- UserA mints 100 shares, and incurs a 2% fee. **98 CoinA** is added to the vault, 100 shares are awarded
- UserB mints 100 shares, and incurs no fee. **100 CoinA** is added to the vault, 100 shares are awarded

The vault holds a total of 198 CoinA and has minted 200 shares.

Upon withdraw (and ignoring further CoinA fees for simplicity):

- UserA has 100 (50%) of the 200 shares and receives 198 CoinA * 100/200= **99 CoinA**
- UserB has 100 (50%) of the 200 shares and receives 198 CoinA * 100/200= **99 CoinA**

UserA lost 1 CoinA, and **UserB gained 1 CoinA**, purely as a result of incorrect share provisioning.

Since investments occur over time similar outcomes occur if the Cellar rebalances among stablecoins with/without fee-on-transfer characteristics.

Also note that fee-on-transfer assets will break ERC4626 requirements that preview functions (e.g. `previewDeposit()`) return shares ≤ the corresponding action's actual amount (e.g. `deposit()`).

Ideally stablecoins which implement fee-on-transfer would be disallowed entirely from the Cellar. This can be achievable via the Sommelier network trust mechanics, but is complicated by token upgradability.

Consider accounting for actual balance increase/decrease and awarding shares accordingly.

[L-01] Deposits can exceed assetDepositLimit and assetLiquidityLimit

~~LOW~~ *Fixed by 3e5be2591201e6d8008ce9f64ccd2cd6cce78715*

In AaveV2StablecoinCellar.sol, take the following code snippet of `maxDeposit()` (the same applies for `maxMint()`):

```
function maxDeposit(address receiver) public view override returns
(uint256 assets) {
```

```
        if (isShutdown) return 0;

        uint256 asssetDepositLimit = depositLimit;
            uint256 asssetLiquidityLimit = liquidityLimit;
        if (asssetDepositLimit == type(uint256).max && asssetLiquidityLimit
    == type(uint256).max)
                return type(uint256).max;

        **uint256 leftUntilDepositLimit =
    asssetDepositLimit.subMinZero(maxWithdraw(receiver));
        uint256 leftUntilLiquidityLimit =
    asssetLiquidityLimit.subMinZero(totalAssets());**

            // Only return the more relevant of the two.
        assets = Math.min(leftUntilDepositLimit, leftUntilLiquidityLimit);
    }
```

If yield is locked and yet to accrue, the value returned by `maxWithdraw(receiver)` or `totalAssets()` is less than the true value. Hence `leftUntilDepositLimit` and `leftUntilLiquidityLimit` would allow users to deposit more than allowed, and once yield accrues, individual deposits and total assets inside the contract will exceed limits in place.

If there is a need to force hard limits, consider doing share value calculation over total balance by not subtracting locked yield.

[L-02] Potential for locked asset deposit due to unsafe typecasting

~~LOW~~ *Fixed by 3e5be2591201e6d8008ce9f64ccd2cd6cce78715*

`enterPosition()` and `exitPosition()` allow for assets above `type(uint240).max` to be deposited unrecoverably.

See the following implementations in AaveV2StablecoinCellar.sol:

```
function enterPosition(uint256 assets) public whenNotShutdown onlyOwner
{
    ERC20 currentPosition = asset;

    totalBalance += uint240(assets);

    _depositIntoPosition(currentPosition, assets);

    emit EnterPosition(address(currentPosition), assets);
}

function exitPosition(uint256 assets) public whenNotShutdown onlyOwner
{
    ERC20 currentPosition = asset;

    totalBalance -= uint240(_withdrawFromPosition(currentPosition,
assets));

    emit ExitPosition(address(currentPosition), assets);
}
```

Consider the following sequence:

1. `enterPosition(uint256 assets)` is invoked once (when `totalBalance == 0`) with an assets value > `type(uint240).max`
2. `exitPosition(uint256 assets)` is invoked at least twice to fully exit the position.

Some subset of assets supplied to `enterPosition()` will remain locked and unrecoverable.

In order for this issue to surface there would need to be a large scale of assets and the specific steps to reproduce are difficult. While the likelihood of this issue surfacing is extremely low, the impact would be high in the case it does.

Consider reverting if assets > `type(uint240).max` for both methods above.

[Q-01] Unused constant

~~CODE QUALITY~~ *Fixed by 3e5be2591201e6d8008ce9f64ccd2cd6cce78715*

In CellarStaking.sol, `UINT_MAX` value is unused. Consider removing this value.

[Info-01] Rewards can remain unutilized

Acknowledged

If total deposits become zero in between reward cycles, then the rewards between the time when total deposits become equal to zero and the time of the next stake will not be utilized.

Consider the following state:

There is a reward cycle and there are two distinct timestamps A and B.

At A, `totalDeposits` have become 0 and at B there is a new stake.

```
|-------------------------reward cycle-----------------------------------|
|------totalDeposits!=0-------|                    |new stake
|---------------------------------A-------------------B-------------------|
```

When someone stakes at B `_updateRewards()` is called and the following happens:

rewardPerTokenStored = rewardPerTokenStored at A (due to totalDeposits==0)

lastAccountingTimestamp = Present time ⇒ B

All the rewards will be calculated from B, hence rewards between A and B are never distributed.

Thies same scenario exists in other versions of staking rewards like Synthetix staking rewards as well.

Even though this edge case may cause some rewards to remain un-utilized it may not be worth resolving this edge case. The likelihood is low because stakers are unlikely to withdraw when rewards are active. Macro felt it was important to make Sommelier aware of this since they have taken extra efforts to make sure all rewards of the reward cycle are utilized via `startProgram`.

Response from Sommelier:

"This is a common issue around many staking contracts and fixing it would require more added complexity to staking/unstaking than benefit. In practice, we don't expect this to happen since during an active reward program there is always an incentive to deposit. If this ever were to occur, any un-utilized rewards could be covered by scheduling a new reward program in `notifyRewardAmount()` to re-distribute them."

[Info-02] Loss of precision in rewardRate calculation

Acknowledged

CellarStaking.sol, line 752 has the following code

```
rewardRate = reward / nextEpochDuration;
```

`rewardRate` is later used to calculate the actual rewards earned, so you should consider that every operation using rewardRate would be imperfect. In a scenario where reward is a small number this would be exceptionally visible.

Consider:

- `nextEpochDuration` = **604800 (one week)**
- `reward` = **(604800 * 2) - 1 = 1209599**
- `rewardRate` = **1209599 / 604800 = 1 (EVM doesn't have floats!)**

If we assume that `rewardRate` is the amount of tokens distributed per second in an epoch, this means that only **604800** tokens will be actually distributed, with a potential of **604799** tokens remaining in the contract.

Response from Sommelier:

"Potential loss of precision has been noted in the natspec, on line 571, since the time of implementation. Furthermore, this is a risk to administrators, not users, and administrators are more likely to be aware of the finer details of how the contract works. In practice, the reward amounts and epoch durations we have discussed would not be affected by loss of precision."

[Info-03] Minted fees dilute investment

Acknowledged

Platform and performance fees are collected by minting shares. This dilutes shareholder value by changing the price of a share and can cause unexpected investor losses during flat performance which would otherwise not have occurred.

For additional information see the corresponding "[Info-03]" Appendix entry.

Response from Sommelier:
"There is no share dilution that occurs after `accrue`. The price of depositor's shares remains the same immediately after accrual, and increases in value over an unlocking period as the expected amount of yield is distributed (eg. if $1000 in yield with a 10% performance fee, expected $900 yield will be distributed to depositors because $100 was taken in shares)."

[Info-04] Incorrect loss realization at redeem/withdraw

Acknowledged

Incremental loss of the underlying asset is only recognized upon calling `accrue()`, resulting in invalid loss realizations upon redeem/withdraw. This creates opportunities for arbitrage at loss.

For additional information see the corresponding "[Info-04]" Appendix entry.

Note: this issue is only a problem in the event of AAVE not being lossless.

Response:

"This cellar's strategy (ie. lending on Aave) is lossless."

[G-01] Packing variables for efficient read and writes

Addressed in 3e5be2591201e6d8008ce9f64ccd2cd6cce78715

Two opportunities exist for packing variables in a more gas-efficient manner: within ICellarStaking.sol, and CellarStaking.sol:

ICellarStaking

```
// Original (3 slots)
struct UserStake {
        uint112 amount;
        uint112 amountWithBoost;
        uint112 rewardPerTokenPaid;
        uint112 rewards;
        uint32 unbondTimestamp;
        Lock lock;
}

// Packed (2 slots)
struct UserStake {
        // slot 0
        uint112 amount; //14
        uint112 amountWithBoost; // 14
        uint32 unbondTimestamp; // 4
        // slot 1
        uint112 rewardPerTokenPaid; // 14
```

```
        uint112 rewards; // 14
        Lock lock; // 1
}
```

CellarStaking

```
// Present: 10 Slots (from L141):
uint256 public override currentEpochDuration;
uint256 public override nextEpochDuration;
uint256 public override rewardsReady;
uint256 public override minimumDeposit;
uint256 public override endTimestamp;
uint256 public override totalDeposits;
uint256 public override totalDepositsWithBoost;
uint256 public override rewardRate;
uint256 public override rewardPerTokenStored;

uint256 private lastAccountingTimestamp = block.timestamp;
```

Most of these variables can be packed if lower data types are acceptable. Specific reductions in type would depend on your risk assumptions.

For ex : `currentEpochDuration` and `nextEpochDuration` if made uint32, it's safe until the year 2106. Please note by that time contract would break anyway since there is typecast at L303 for `block.timestamp` to `uint32`.

[G-02] UnstakeAll (and all *All methods) that write to storage can be condensed

Acknowledged

`CellarStaking.unstakeAll()` iterates on all the stakes and calls `_unstake()`, which writes to storage for each stake. `CellarStaking._unstake()` has the following code:

```
// Update global state
totalDeposits -= depositAmount;
totalDepositsWithBoost -= amountWithBoost;

// Distribute stake
stakingToken.safeTransfer(msg.sender, depositAmount);

// Distribute reward
distributionToken.safeTransfer(msg.sender, reward);
```

Consider breaking up the unstaking process in a way that the changes to storage including `safeTransfer` calls will be done once for each storage variable.

Also notice the tests for `*All()` methods only operate on a few stakes, consider adding more stakes to better spot excessive gas usage.

Similar pattern observed in:

- `emergencyUnstake`
- `claimAll`
- `unbondAll`
- `cancelUnbondingAll`

Response from Sommelier :

"While these savings are significant, given the late-stage nature of this contract and short time to release, this represents more of a change to logic than we are comfortable with (would touch many different lines of code). This can be revisited for future cellars, after our initial release."

[G-03] Cache latestRewardsTimestamp in _updateRewards()

Addressed in 3e5be2591201e6d8008ce9f64ccd2cd6cce78715

Within CellarStaking.sol:

```
function _updateRewards() internal {
    rewardPerTokenStored = rewardPerToken();
    lastAccountingTimestamp = latestRewardsTimestamp();
}

function rewardPerToken() public view override returns (uint256) {
    if (totalDeposits == 0) return rewardPerTokenStored;

    uint256 timeElapsed = latestRewardsTimestamp() -
lastAccountingTimestamp;
    uint256 rewardsForTime = timeElapsed * rewardRate;
    uint256 newRewardsPerToken = (rewardsForTime * ONE) /
totalDepositsWithBoost;

    return rewardPerTokenStored + newRewardsPerToken;
}

function latestRewardsTimestamp() public view override returns
(uint256) {
    return block.timestamp < endTimestamp ? block.timestamp :
endTimestamp;
}
```

Please check `latestRewardsTimestamp()`.

`latestRewardsTimestamp` is being executed twice in `_updateRewards()`, once in `rewardPerToken()` and once in itself.

By returning the `latestRewardsTimestamp` from `rewardPerToken()` to `_updateRewards()`, this second execution can be avoided.

`_updateRewards()` is a heavily used function throughout the staking contract, so each save is valuable.

[G-04] Redundant check in stake()

Acknowledged

The check on CellarStaking.sol L216, becomes redundant due to L217

```
L216: if (amount == 0) revert USR_ZeroDeposit();
L217: if (amount < minimumDeposit) revert USR_MinimumDeposit(amount,
minimumDeposit);
```

Response from Sommelier:
"Removing the check to the contract not reverting when `minimumDeposit == 0`. We want deposits to revert in that case, without having to explicitly set a non-zero minimum deposit in the constructor."

[G-05] Redundant operation in _cancelUnbonding()

Addressed in 3e5be2591201e6d8008ce9f64ccd2cd6cce78715

The logic from CellarStaking.sol L357-358 can be written such that there is one less operation.

```
// Original (1 ADD 1 MUL 1 DIV 1 SUB, 3 SLOAD )
uint256 amountWithBoost = s.amount + (s.amount * boost) / ONE;
uint256 depositAmountIncreased = amountWithBoost - s.amountWithBoost;

// Optimized (1 MUL 1 DIV 1 ADD, 2 SLOAD)
uint256 depositAmountIncreased = (s.amount * boost) / ONE;
```

```
uint256 amountWithBoost = s.amount + depositAmountIncreased

// DIFF: 1 SUB 1 SLOAD
```

[G-06] Redundant variable in _unstake()

Addressed in 3e5be2591201e6d8008ce9f64ccd2cd6cce78715

In unstake `depositAmount/s.amount` and `amountWithBoost/s.amountWithBoost` are always going to be the same. So if you are reading `s.amount` at L417, and storing it in `depositAmount`, you don't have to read and store `s.amountWithBoost` separately on L425.

Saving 1 SLOAD

[G-07] Redundant variable read in emergencyStop()

Addressed in 3e5be2591201e6d8008ce9f64ccd2cd6cce78715

In CellarStaking.sol `emergencyStop():claimable` is already present in a memory variable via `makeRewardsClaimable`, hence you can use that on L653, instead of `claimable`.

Saving 1 SLOAD

[G-08] Redundant normalization in convertToAssets() and convertToShares()

Addressed in 3e5be2591201e6d8008ce9f64ccd2cd6cce78715

In AaveV2StablecoinCellar.sol, there is redundant normalization within `convertToAssets()` and `convertToShares()` which can be avoided to reduce gas costs. See `convertToAssets()`:

```
function convertToAssets(uint256 shares) public view
override returns (uint256 assets) {
    // L385:
    uint256 totalShares = totalSupply; // Saves an extra
    SLOAD if totalSupply is non-zero.
    uint8 positionDecimals = assetDecimals;
    uint256 totalAssetsNormalized =
    totalAssets().changeDecimals(positionDecimals, 18);
    // L389:
    assets = totalShares == 0 ? shares :
    shares.mulDivDown(totalAssetsNormalized, totalShares);
    assets = assets.changeDecimals(18, positionDecimals);
}
```

On 388, `totalAssets` are being normalized On 390, if `totalShares!=0`, the amount of assets needed for this many shares is being calculated. **Please check this arithmetic.** Since shares are being divided by `totalShares`, the resultant will be in decimals of assets only. Hence if you didn't normalize on L388 to 18 decimals, you don't need to normalize again at L391 to `positionDecimals`. The explicit normalization is only needed when `totalShares == 0`, as shares are in 18 decimals.

The block L388-L391, can be rewritten as

```
if(totalShares==0) assets = shares.changeDecimals(18,
positionDecimals);
else assets = shares.mulDivDown(totalAssets(), totalShares);
```

Automated Analysis

Slither

Slither is a solidity static analysis framework. It detects many vulnerabilities, from high threats to benign ones, of which there are usually many.

In order to run Slither against the codebase we used the following configuration file and ran the following commands:

- slither.config.json

```
{
  "filter_paths": "lib",
  "solc_remaps": [
    "@solmate/=./lib/solmate/src/",
    "@forge-std/=./lib/forge-std/src/",
    "@ds-test/=./lib/forge-std/lib/ds-test/src/",
    "@openzeppelin/=./lib/openzeppelin-contracts/contracts/",
    "@uniswap/v3-periphery/=./lib/v3-periphery/contracts/",
    "@uniswap/v3-core/=./lib/v3-core/"
  ]
}
```

- `$ slither ./src/AaveV2StablecoinCellar.sol --solc-args "--optimize --optimize-runs 200"`
- `$ slither ./src/CellarStaking.sol --solc-args "--optimize --optimize-runs 200"`

Slither identified many issues; manual inspection and a detailed understanding of both the intended behavior and implementation revealed that nearly all of the issues identified by slither were false positives. However, [G-02] has been confirmed as an issue.

Appendix - additional context

Additional context and information for reported issues.

[M-01] Inaccurate platform fee accounting

The function name (`accrue()`), the variable naming (`balanceLastAccrual`, `balanceThisAccrual`), and related comments sends the signal that `totalBalance` is tracked accrual-to-accrual. However, a number of functions change the value of `totalBalance` between accruals:

- `beforeWithdraw()` line 343
- `enterPosition()` line 534
- `exitPosition()` line 555
- `rebalance()` line 640
- `emptyPosition()` line 793

This opens the door for platform fee accounting to yield inaccurate fees.

Consider Scenario A:

- Block A
  - `accrue()` is called
- Some period of time passes
- Block M
  - Alice `deposit()`'s assets
  - `enterPosition()` called, which increases `totalBalance`
  - `accrue()` called
- Some period of time passes
- Block X

    - `accrue()` is called

Because Alice's deposit and updated position happened at Block M, it is expected that platform fees against this deposit would only be charged for the period between **Block M** and Block X.

Instead, the above accounting logic calculates platform fees for Alice's deposit from **Block A** through Block X. This happens because `enterPosition()` updated `totalBalance` prior to `accrue()`, which treats this new higher balance as having existed since Block A's `lastAccrual`.

As a result, **more** platform fees than are warranted get extracted.

Consider Scenario B:

Initial state: Alice has a large deposit, which is fully entered into position.

- Block A
    - `accrue()` is called
- Some period of time passes
- Block M
    - Alice withdraws all assets via `withdraw()`, which reduces `totalBalance`
    - `accrue()` called
- Some period of time passes
- Block X
    - `accrue()` is called

Because Alice's withdrawal happened at Block M, it is expected that platform fees would be charged for Alice's deposit, which existed between Block A and Block M.

Instead, the above accounting logic calculates platform fees *without* Alice's deposit during this time. This occurs because `withdraw()` updated `totalBalance` prior to `accrue()`, which treats this new lower balance as having existed since Block A's `lastAccrual`.

As a result, **less** platform fees than are warranted get extracted.

[Info-03] Minted fees dilute investment

See the `accrue()` function, lines 511-512:

```
// Mint accrued fees as shares.
_mint(address(this), platformFees + performanceFees);
```

Minting shares alone dilutes the share-to-asset ratio, which can in certain circumstances dilute user investments in unexpected ways. Consider the following example:

1. Alice invests 1000 USDC. Position entered.
    - Cellar has 1000 shares and 1000 aUSDC
    - 1 share = 1.0000 USDC
2. Yield increases by 100 aUSDC (10% gain)
    - Cellar has 1000 shares and 1100 aUSDC
    - 1 share = 1.0500 USDC
3. Bob invests 1000 USDC. Position entered.
    - Cellar has 2000 USDC and 2100 aUSDC
    - 1 share = 1.0500 USDC
4. `accrue()` and then wait for the accrual period to end
    - 10 shares in performance fees are minted
    - Cellar has 2010 shares and 2100 aUSDC
    - 1 share ~= 1.0448 USDC

The operation of minting shares dilutes share price by changing the share-to-asset ratio, such that investors will lose some amount of money regardless of the actual performance of their personal investment. In this case, Bob's investment saw none of the gain because it arrived after the yield, but Bob incurred a loss anyway. Bob would not have incurred an unnecessary loss if the fees were extracted in a manner that did not change the share-to-asset ratio.

[Info-04] Incorrect loss realization at redeem/withdraw

The true-up of actual asset balance only occurs with calls to `accrue()`. This means that the underlying asset can fluctuate in either direction, and deposit/withdraw actions by users will occur according to the previously "pinned" balance number.

Note that the `rebalance()` function performs a similar check, but the focus of this issue is isolated to a single persistent asset's gain/loss.

The asset balance is check within `accrue()` on line 499:

```
uint256 balanceThisAccrual = assetAToken.balanceOf(address(this));
```

and then the cellar total balance is updated on line 519:

```
totalBalance = uint240(balanceThisAccrual);
```

Consider the following scenario:

1. Initial state: Alice & Bob deposited 1000 USD each. Position entered.
    1. Cellar has 2000 USDC and 2000 shares
2. Performance loss of 50% occurs
    1. Cellar has 2000 USDC and 2000 shares (it should have 1000 USDC, but `accrue()` was not called to update the balance)
3. Alice redeems 1000 shares
    1. Cellar has 1000 USDC and 1000 shares (again, cellar accounting is still off)
    2. **Alice receives $1000 USDC**

4. `accrue()`
    1. Cellar has 500 USDC and 1000 shares
5. Bob redeems 1000 shares
    1. Cellar has 0 USDC and 0 shares
    2. **Bob receives $500 USDC**

In this scenario, Alice and Bob both redeem their shares during the period when the asset value has dropped by 50%. They should both receive $500 USDC. However, the loss of the underlying asset is only realized by the Cellar at the call to `accrue()`, so Alice redeems at

the previously “pinned” value of $1 USDC per share. Bob redeems at the newly “pinned” value of $0.5 USDC per share.

Additionally, Alice can re-deposit her $1000 USDC immediately after the `accrue()`, earning 2000 shares.

As a result, Alice’s shares have increased during a performance loss when they shouldn’t have.

Disclaimer

Macro makes no warranties, either express, implied, statutory, or otherwise, with respect to the services or deliverables provided in this report, and Macro specifically disclaims all implied warranties of merchantability, fitness for a particular purpose, noninfringement and those arising from a course of dealing, usage or trade with respect thereto, and all such warranties are hereby excluded to the fullest extent permitted by law.

Macro will not be liable for any lost profits, business, contracts, revenue, goodwill, production, anticipated savings, loss of data, or costs of procurement of substitute goods or services or for any claim or demand by any other party. In no event will Macro be liable for consequential, incidental, special, indirect, or exemplary damages arising out of this agreement or any work statement, however caused and (to the fullest extent permitted by law) under any theory of liability (including negligence), even if Macro has been advised of the possibility of such damages.

The scope of this report and review is limited to a review of only the code presented by the Emergent team and only the source code Macro notes as being within the scope of Macro’s review within this report. This report does not include an audit of the deployment scripts used to deploy the Solidity contracts in the repository corresponding to this audit. Specifically, for the avoidance of doubt, this report does not constitute investment advice, is not intended to be relied upon as investment advice, is not an endorsement of this project or team, and it is not a guarantee as to the absolute security of the project. In this report you may through hypertext or other computer links, gain access to websites

operated by persons other than Macro. Such hyperlinks are provided for your reference and convenience only, and are the exclusive responsibility of such websites' owners. You agree that Macro is not responsible for the content or operation of such websites, and that Macro shall have no liability to your or any other person or entity for the use of third party websites. Macro assumes no responsibility for the use of third party software and shall have no liability whatsoever to any person or entity for the accuracy or completeness of any outcome generated by such software.