yAudit TempleDAO Lending Review

Review Resources:

- Existing TempleDAO docs
- Figma diagrams of in-scope contracts call flows:
  - High level overview
  - RAMOS
  - TLC
  - Gnosis Safe Strategy
  - Circuit Breakers

Auditors:

- engn33r
- adriro

Table of Contents

Review Summary

TempleDAO

TempleDAO provides investors with a protocol that aims to provide a stable form of value and investment. The protocol design is centered around the TEMPLE token, backed by the TempleDAO treasury. The protocol has various mechanisms in place to manage risk while growing the treasury and in turn increasing the value of the TEMPLE token. The code in this review centered around two key components of the protocol:

1. The strategy architecture, which allows the Treasury Reserves Vault (TRV) to invest assets to generate yield that accrues to the treasury.
2. The RAMOS AMO, which provides price stabilization by managing the TEMPLE-DAI Balancer pool.

The contracts of the TempleDAO Repo were reviewed over 24 days. The code review was performed by 2 auditors between July 10 and August 3, 2023. 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 b015cd5d1df122ad5fbe0f94fb5bd070db27e335 for the TempleDAO repo, and the review scope was focused on the v2 and amo directories.

Scope

The scope of the review consisted of the following contracts at the specific commit:

- https://github.com/TempleDAO/temple/tree/audit-v2/protocol/contracts/v2/*
- https://github.com/TempleDAO/temple/tree/audit-v2/protocol/contracts/amo/*

Note that the deployment scripts were not in a finalized state at the commit hash used for the audit, so some code and the values set in the deployment scripts at the reviewed commit hash were expected to differ from the final on-chain values. After the findings were presented to the TempleDAO 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.

yAudit and the auditors make no warranties regarding the security of the code and do not warrant that the code is free from defects. yAudit and the auditors 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, TempleDAO and users of the contracts agree to use the code at their own risk.

Code Evaluation Matrix

Category	Mark	Description
Access Control	Good	The majority of functions have an access modifier such as `onlyElevatedAccess`. There are only two main roles which are assigned to governance multisigs, executor and rescuer.
Mathematics	Average	Some complexity was involved in the compound interest calculations, especially in DsrBaseStrategy and CompoundedInterest. Otherwise, the math found in the contracts was only involved in standard accounting logic.
Complexity	Average	The combination of a compounding debt token and multiple strategies pulling from a single vault increases the complexity of the system's accounting logic beyond a

Category	Mark	Description
		basic DeFi protocol, but only to the level of an average protocol's complexity.
Libraries	Average	OpenZeppelin contracts, PRB math, and Gnosis Safe contracts were the main imports. These libraries are considered reputable, but external dependencies do add a risk vector for introducing vulnerabilities.
Decentralization	Low	TempleDAO relies heavily on trusted governance to perform many key actions, including transferring tokens to arbitrary recipient addresses. While such actions require multiple multisig signers for approval, the design does not prioritize decentralization. The stablecoin trilemma is a trade-off between decentralization, price stability, and capital efficiency, so the TempleDAO design makes it clear what is being optimized for.
Code stability	Good	All code examined in the review was in a polished and nearly production ready state. The RAMOS AMO code is already in production, albeit an older version of the contract.
Documentation	Good	All important external functions had NatSpec, though some internal functions with relatively clear purposes didn't have NatSpec. Clear architecture diagrams were created demonstrating the interactions between different contracts.
Monitoring	Good	Besides one case mentioned in a finding, events are emitted in most functions that involve value transfer.
Testing and verification	Good	The in-scope contracts all have good test coverage with at or near 100% test coverage. The amo contracts use hardhat tests while the v2 contracts use foundry tests.

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

1. High - Debtor debt can be cleared if rate is set to zero

The debt accrued to a debtor in the TempleDebtToken.sol contract may be unintentionally cleared if the risk premium interest rate is set to zero.

Technical Details

Debtor interests are calculated using the function `_compoundedDebtorInterest()`. The implementation of this function will return zero if the rate is set to zero:

```
function _compoundedDebtorInterest(Debtor storage debtor) internal view returns (uint256)
{
    uint256 _rate = debtor.rate;
    if (_rate == 0) return 0;
    ....
```

If the interest rate is first set to a positive number and then later set to zero, then the accrued debt of the debtor will be nullified as it will be reset to zero. As we can see in the following snippet of code, the return value of `_compoundedDebtorInterest()` will be stored in `debtor.checkpoint`, which represents the current accrued debt.

```
function _checkpointDebtor(Debtor storage debtor) internal returns (uint256) {
    uint256 interest = _compoundedDebtorInterest(debtor);
    unchecked {
        estimatedTotalRiskPremiumInterest += (interest - debtor.checkpoint);
    }
    debtor.checkpoint = uint160(interest);
    debtor.checkpointTime = uint32(block.timestamp);
    return interest;
}
```

Note also that `balanceOf()` and `currentDebtOf()` will be affected as these use `_compoundedDebtorInterest()` internally.

Impact

High. Any debtor's risk premium interest can be unintentionally cleared.

Recommendation

When the rate is zero, return the current debt instead of zero.

```
function _compoundedDebtorInterest(Debtor storage debtor) internal view returns (uint256)
{
    uint256 _rate = debtor.rate;
    if (_rate == 0) return debtor.checkpoint;

    uint256 _timeElapsed = block.timestamp - debtor.checkpointTime;
    uint256 _principal = debtor.principal;
    uint256 _principalAndInterest = _principal + debtor.checkpoint;
    return _principalAndInterest.continuouslyCompounded(_timeElapsed, uint96(_rate)) -
_principal;
}
```

Developer Response

Recommendation implemented and tests added in PR #842

Medium Findings

1. Medium - TPI can still be updated while there is a pending change

Updates to the treasury price index in TreasuryPriceIndexOracle.sol are delayed by a cooldown period in which the oracle will continue to output the old value. However, it is still possible to update the price while there is a pending change, which will nullify the effect of the cooldown.

Technical Details

The TreasuryPriceIndexOracle.sol uses a cooldown mechanism for price updates. Whenever the price is updated using `setTreasuryPriceIndex()`, the contract will store the overwritten value and will continue to return this price as long as the cooldown duration is in effect.

However, there's nothing preventing this function from being called again during the cooldown period. Doing so will nullify the cooldown for the current change: the `currentTpi` value (which should be active after the cooldown) will become the `previousTpi` and `treasuryPriceIndex()` will return this value before the original cooldown expires.

After discussions with the devs, it was determined that the cooldown is intended to allow for the TPI to be changed again during the cooldown period if the first TPI value was set incorrectly. The logic flow in this case where `setTreasuryPriceIndex()` is called a 2nd time during the cooldown period to fix an error causes an even larger problem. The problem is that during the cooldown period, the previous TPI value is considered the correct TPI value. If an incorrect TPI value is set, it will be stored in the `currentTpi` variable, but when it gets replaces with a second `setTreasuryPriceIndex()` call, the value previously stored in `currentTpi` will be stored as `previousTpi`. This means the incorrect TPI value will immediately become the real TPI value used by the protocol during the process of attempting to replace and fix the original mistake. The result is that the cooldown and the surrounding logic potentially make the situation worse than if the contract didn't implement the cooldown, because in the current flow, an attempt to fix a mistake by updating the TPI causes the mistaken TPI value to immediately be used in production.

Note also that it is possible to adjust the cooldown duration while there is a current pending change too. This is another way that the TPI can be changed faster than the existing cooldown.

Impact

Medium. While this function requires privileged access, both scenarios described can lead to high impact consequences.

Recommendation

Instead of updating the previous price (`tpiData.previousTpi`) using the value of the current price (`tpiData.currentTpi`), if there is a cooldown in effect, just edit the current price using the new value, leaving the same value for the previous price. This will allow changes while still keeping the cooldown.

```
  function setTreasuryPriceIndex(uint256 value) external override onlyElevatedAccess {
-     uint96 _oldTpi = tpiData.currentTpi;
+     uint96 _oldTpi = block.timestamp < (tpiData.lastUpdatedAt + tpiData.cooldownSecs) ?
tpiData.previousTpi : tpiData.currentTpi;
      uint96 _newTpi = value.encodeUInt96();

      uint256 _delta = (_newTpi > _oldTpi) ? _newTpi - _oldTpi : _oldTpi - _newTpi;
      if (_delta > maxTreasuryPriceIndexDelta) revert BreachedMaxTpiDelta(_oldTpi,
_newTpi, maxTreasuryPriceIndexDelta);

      emit TreasuryPriceIndexSet(_oldTpi, _newTpi);

      tpiData = TpiData({
          currentTpi: _newTpi,
          previousTpi: _oldTpi,
          lastUpdatedAt: uint32(block.timestamp),
          cooldownSecs: tpiData.cooldownSecs
      });
  }
```

Developer Response

Recommendation implemented and tests added in PR #842

2. Medium - Updating the debt ceiling for the line of credit strategy should trigger an update to the interest rate

The interest rate of the TempleLineOfCredit.sol contract depends on the utilization ratio, which is defined to be the total debt over the corresponding strategy debt ceiling. This rate should be updated whenever the debt ceiling corresponding to the TlcStrategy.sol contract gets updated.

Technical Details

The calculation of the interest rate in the line of credit contract depends on the interest rate model and the utilization ratio, which is the debt over the available amount of tokens, defined as the debt ceiling of the strategy. If any of these values changes, the interest rate should be updated.

While the rate is updated when the strategy changes, there is nothing ensuring that the interest rate is recalculated if the debt ceiling for the line of credit strategy gets updated in the TreasuryReservesVault.sol contract.

Impact

Medium. The line of credit interest rate may not be in sync with the proper configuration if the strategy's debt ceiling is updated in the vault contract.

Recommendation

The current architecture doesn't offer a way to notify the strategy whenever its debt ceiling is updated. This could be implemented by adding a hook in AbstractStrategy.sol that gets triggered when this parameter is updated. TlcStrategy.sol can then refresh the interest rate when this callback gets triggered.

Alternatively, the protocol should ensure to manually call `refreshInterestRates()` as part of the update to the debt ceiling of TlcStrategy.sol.

Developer Response

The recommendation of a hook was implemented and tests added in PR #842

3. Medium - Possibility of TLC bad debt from TPI increase

TempleLineOfCredit.sol allows lending of DAI with TEMPLE collateral and prices TEMPLE based on Treasury Price Index (TPI). TPI is a variable set by TempleDAO governance. Meanwhile, the actual spot price of TEMPLE is determined by the TEMPLE-DAI Balancer pool. If the difference between TPI and instantaneous price is large, the TLC may accumulate some bad debt. This bad debt would later lower the TPI, reducing the price of the TEMPLE token.

Technical Details

The TPI is a value that is set by TempleDAO governance. This value is updated in a single block. For example, the latest TPI increase can be seen with:

```
cast call 0xC3133cB9e685ccc82C73FbE580eCeDC667B41917 "templePriceFloorNumerator()
```

```
(uint256)" --block 17737310
```

Return Value: 10250

```
cast call 0xC3133cB9e685ccc82C73FbE580eCeDC667B41917 "templePriceFloorNumerator()
(uint256)" --block 17737311
```

Return Value: 10600

The TPI value is assumed to be equal to the price of TEMPLE when the value of TEMPLE collateral is calculated in TempleLineOfCredit.sol. This is in contrast to lending protocols such as Compound Finance, which use the current price of a token from Chainlink to determine the value of user collateral. It is possible, in fact likely, that the TPI value is greater than the value of TEMPLE based on the spot price of the Balancer pool. The graph illustrating the variation of the two prices is found on TempleDAO's homepage.

If the difference between the spot price in the Balancer pool and the TLC is too great, an arbitrageur can leave the protocol with bad debt. Consider a scenario where TPI was at $1, the spot price of TEMPLE is $0.90, and the TPI was just raised by governance to $1.15:

1. User takes DAI flashloan from Balancer with a 0% cost.
2. User swaps DAI for TEMPLE in TEMPLE-DAI balancer pool. The volume of the swap can raise the TEMPLE price in the pool to $0.95
3. User deposits the newly acquired TEMPLE into the TLC and maxes out their LTC with a 85% LTV loan.
4. The user will get liquidated almost immediately because the interest accumulated in the next block will cause them to exceed the max LTC threshold of 85%. But the user doesn't care, because $1 worth of DAI got them 1/0.95 = 1.053 TEMPLE. This TEMPLE is valued at 1.053*1.15 = $1.211 and can have 85% of its value borrowed, which is 1.211*0.85 = 1.029
5. In this example scenario, the arbitrageur walks away with a 2.9% profit. The bad debt will lower the holdings of the TempleDAO treasury and reduce the TPI, impacting TEMPLE token price.

Another version of this bad debt scenario can happen in reverse if the TPI is decreased significantly while the TEMPLE spot price is significantly below TPI:

1. User takes TEMPLE flashloan from Balancer with a 0% cost.
2. User deposits TEMPLE into TLC, takes out DAI loan with max LTV when TPI is high relative to current TEMPLE token price.
3. User swaps DAI to TEMPLE in Balancer pool when TEMPLE price is below TPI.
4. User repays flashloan, retaining any profit. The profit comes at the expense of the TempleDAO treasury.

There are already some protective measures preventing this scenario, but because these protective measures are controlled by governance, it's possible that future protocol changes may limit the effectiveness of these protective measures. The protective measures include:

1. RAMOS should rebalance the price of TEMPLE in the Balancer pool if the price of TEMPLE drops below 1% of the TPI (note: the script linked to was not in the commit hash that the audit focused on).
2. The `maxTreasuryPriceIndexDelta` value limits how large of a change the TPI can experience thanks to a check in `setTreasuryPriceIndex()`. The current choice of setting `maxTreasuryPriceIndexDelta` to 0.05 combined with the current choice of setting the max LTV to 0.85 should protect the protocol. However, be aware these values can be changed by governance, and there is no cooldown on `setTreasuryPriceIndex()`, meaning there is no real limit to how much governance can increase TPI in a single block.

The first protective measure mentioned, the RAMOS rebalancing, is run by a bot controlled by governance. This bot is running off-chain and has the potential for more downtime that the Ethereum blockchain. Therefore, when consider extreme edge cases, the most extreme case is a scenario where the RAMOS bot is offline and is temporarily unable to stabilize the price of TEMPLE in the Balancer pool.

Impact

Medium. In extreme market conditions, bad debt can accumulate to TempleDAO if the TPI is raised too much while the token price is down.

Recommendation

Some additional mitigations can reduce the risk of this issue:

1. Adding a cooldown to `setTreasuryPriceIndex()` can avoid a scenario where TPI is updated too quickly without giving time for RAMOS to increase and stabilize the token price to the new TPI value.

2. Because TPI is set manually, consider adding a step in the governance process to confirm that the difference between the new TPI and the current token price is less than a certain threshold before increasing TPI, for example 10%. A similar check should be added when the TPI is going to be lowered, but the current TPI (not the new TPI) should be compared to the token price.

Developer Response

We agree that a large delta between the TPI and the spot price is a scenario that can lead to bad debt on the Temple Line of Credit. We have controls in place to mitigate the odds of this scenario:

- RAMOS is designed to keep price within 3% of the TPI
- TPI increases are manually executed and not updated automatically based on an oracle
- Change to TPI requires TempleElevatedAccess
- maxTreasuryPriceIndexDelta limits the TPI change to no more more than 5% per update
- TLC can be put into PAUSE if we have reason to believe the TPI cannot be ascertained with accuracy, or if the spot price of $TEMPLE cannot be maintained by the bot.

We do not necessarily agree that a cooldown or timelock for TPI changes mitigates the issue because of the risk of toxic flow front-running TPI update (e.g. borrow more before TPI goes down or buy more TEMPLE before TPI goes up). Ideally there would be a drip () that moves the TPI gradually between Treasury updates. But the current composition of active Temple Strategies makes that approach infeasible due to unreliable and/or unknown value of current holdings. For instance, a different Treasury-backed token or stablecoin may be trading below its backing value (analogous to TPI). It is unclear when/how to update Temple's TPI in an automated way that marks to market or marks to theoretical book value of those holdings (e.g. pegged or soft-pegged value).

4. Medium - `checkNSignatures()` logic causes revert

`SafeForked.checkNSignatures()` has an extra line of code not found in the original Gnosis Safe code that will always revert. This revert will happen on every multisig transaction that relies on the custom Gnosis guard, requiring a redeployment of the code to fix the issue.

Technical Details

`SafeForked.checkNSignatures()` has an extra line `require(keccak256(data) == dataHash, "GS027");` that is not found in v1.3.0. This line is found in the latest v1.4.1 release of the Gnosis

Safe code. But the values of `dataHash` and `data` in the gnosis code are set differently than in the TempleDAO code. When `checkNSignatures()` is called from `ThresholdSafeGuard.checkTransaction()` with the line `SafeForked.checkNSignatures(safeTxExecutor, safe, keccak256(txHashData), data, signatures, threshold);`, the expectation is that `data == txHashData` to pass the later require statement. But `data != txHashData` because txHashData consists of encoded values that includes `data`.

If this `require()` check fails, all calls to `ThresholdSafeGuard.checkTransaction()` that include at least one contract signature (when `v == 0`) will fail, reverting the transactions passing through the Gnosis guard.

After discussions with the development team, the bug exists in the if/else branch where `v == 0`. This logical branch is not expected to be reached because the new governance multisig that will be using the Gnosis Safe Guard will only have EOA signers and no smart contract signers. This means the problematic line of code should not be reached by TempleDAO logic.

Impact

Medium. Code reverts due to a logic bug in a specific logic flow, but developers state that this flow will not be used by the multisig per current design intentions.

Recommendation

The 4th argument in the call to `checkNSignatures()` should be `txHashData` instead of `data`. This is what is done in the original Gnosis contract.

Developer Response

Recommendation implemented and tests added in PR #842

Low Findings

1. Low - Can repay debt for wrong borrow token

The Treasury Reserves Vault (TRV) has two valid borrow tokens, DAI and TEMPLE. It is possible for anyone to call the `repay()` or `repayAll()` functions for a strategy to pay off the strategy's debt. If a strategy supports only one token, DAI or TEMPLE, it is possible to pass the other token as a function argument to `repay()` and boost the credit of this strategy in an unsupported token. Although no clear attack vector was identified here and the caller would still have the transfer tokens, future changes to TempleDAO logic may allow this to cause accounting issues.

Technical Details

`repay()` and `repayAll()` accept a `token` and `strategy` function argument. There are two valid `token` values, the DAI address or the TEMPLE address. This is confirmed by the existing strategies and the deployment scripts for dUSD and dTEMPLE as the only debt tokens. There is no verification that the `token` and `strategy` arguments match, meaning that the `token` address can be a borrow token that does not match with the borrow token used by the `strategy`. The only strategy that handles more than one token (DAI and TEMPLE) is the Ramos strategy, so the other strategies should be limited to accumulating only debt in the token that they handle.

It is unclear how the debt token and credit balances are queried off-chain to handle accounting of the overall system (this accounting is not performed on-chain with an equivalent of yearn vault's `report()`), but this scenario may cause accounting issues if the debt and credit balances are handled improperly. The caller of `repay()` does donate tokens to the protocol, so there is a cost to this call flow.

A foundry test demonstrating this issue, based on the existing `test_repay_withDTokenDebts_noCreditLeft` test:

```
    function test_repay_withDTokenDebts_wrongToken() public {
        // Setup the config
        {
            vm.startPrank(executor);
            trv.setBorrowToken(dai, address(0), 0, 0, address(dUSD));
            trv.setBorrowToken(temple, address(0), 0, 0, address(dTEMPLE));
            deal(address(dai), address(trv), 120e18, true);
            deal(address(temple), address(trv), 120e18, true);

            ITempleStrategy.AssetBalance[] memory debtCeiling = new
ITempleStrategy.AssetBalance[](1);
            debtCeiling[0] = ITempleStrategy.AssetBalance(address(dai), 50e18);
            trv.addStrategy(address(strategy), -123, debtCeiling);
        }

        // Borrow some so there's a debt
        {
```

```
            changePrank(address(strategy));
            trv.borrow(dai, 5e18, address(strategy));
        }

        // Fund alice so she can repay on behalf of the strategy
        {
            deal(address(temple), alice, 50e18, true);
            changePrank(address(alice));
            temple.approve(address(trv), 50e18);
        }

        {
            trv.repay(temple, 7e18, address(strategy));
        }

        // TEMPLE transferred to TRV, dToken DAI credit given to strategy as it started
with no debt
        assertEq(dai.balanceOf(address(strategy)), 5e18);
        assertEq(dai.balanceOf(address(trv)), 120e18 - 5e18);
        assertEq(temple.balanceOf(address(alice)), 50e18-7e18);
        assertEq(trv.strategyTokenCredits(address(strategy), dai), 0);
        assertEq(trv.strategyTokenCredits(address(strategy), temple), 7e18);
        assertEq(dUSD.balanceOf(address(strategy)), 5e18);
        assertEq(dTEMPLE.balanceOf(address(strategy)), 0);
    }
```

Impact

Low. There is potential for accounting issues due to the unexpected code paths and how it impacts internal debt/credit values.

Recommendation

Consider adding a public `supportedTokens` array to AbstractStrategy.sol so that the user-specified `token` argument in `repay()` and `repayAll()` can be checked against.

Developer Response

Recommendation implemented and tests added in PR #842

2. Low - Potential clashing between function calls and ETH payments in ThresholdSafeGuard.sol

ThresholdSafeGuard.sol implements a Gnosis Safe Guard that checks for a defined threshold on a per function configuration basis. The implementation contains a wrong assumption that could potentially cause a conflict between certain function calls and ETH payments.

Technical Details

Threshold configurations for functions are stored in a mapping whose keys are the contract address and the function selector. The check then queries this mapping to fetch the associated threshold setting.

ETH transfer transactions are represented by a function selector of zero, as indicated by this comment:

> @dev functionSignature=bytes(0) is ok as this represents an ETH transfer which may also have an explicit threshold.

As zero function selectors are valid (see issue *Function selectors with a zero value are valid*), this unveils a potential clashing between a function whose selector is zero and an ETH transfer. The `checkTransaction()` function casts the calldata parameter to `bytes4` without checking the size:

```
...
threshold = getThreshold(to, bytes4(data));
...
```

If the `data` argument is empty, then `bytes4(data)` will be zero, creating a conflict with a potential call to a function with a zero selector.

Impact

Low. Even though a zero function selector is valid, the probability of a conflict would be low.

Recommendation

Check the size of the `data` argument to discriminate between ETH payments and a potential valid call to a function whose selector is zero.

Developer Response

Recommendation implemented and tests added in PR #842

To set a threshold specific to ETH payments, a new `setEthTransferThreshold()` must be called. The Guard now checks if `data` is empty as suggested, and if so uses this new specific threshold.

3. Low - `currentUtilisation()` can return stale data

In a scenario where `preCheck()` has not been called for a while, and therefore the bucket amounts are stale, the value returned by `currentUtilisation()` can be inaccurate. `_currentUtilisation()` assumes all bucket values are from the most recent `nBuckets`, but this assumption is invalid when called from the external `currentUtilisation()`.

Technical Details

The logic of `_currentUtilisation()` is to loop through the buckets and sums the amounts. But this naive approach omits the possibility that the buckets contain old data that will be cleared on the next `preCheck()` and therefore the values stored in some buckets may not be relevant to checking the proximity to the current cap. These stale buckets should be ignored, the same way those stale buckets are cleared in `preCheck()`. This is only an issue when `_currentUtilisation()` is called from `currentUtilisation()`, because when called from `preCheck()` the buckets are updated before the call to `_currentUtilisation()`.

Impact

Low. `currentUtilisation()` can return inaccurate data.

Recommendation

Copy the logic from `preCheck()` into `_currentUtilisation()` to ignore stale buckets, removing the lines of code that modify state variables.

Developer Response

Recommendation implemented and tests added in PR #842

4. Low - `setConfig()` resets buckets and gives minor circuit breaker bypass

If there is an action that a user wanted to take with TempleDAO that would normally be blocked by the circuit breaker, the best time to perform such an action is by frontrunning `setConfig()` because the bucket values will be reset after `setConfig()` is called.

Technical Details

`setConfig()` lets governance choose new values for period duration, buckets, and cap. In this process, it resets the amount value of every bucket. While the bucket values are all reset to zero (or 1, though this dust is subtracted out later), this gives a window of opportunity to

maximize the value passing through the circuit breaker. The worst case scenario with one `setConfig()` call is twice the circuit breaker cap value, and this can be generalized to n+1 times the circuit breaker cap for n number of `setConfig()` calls in a short timespan. So if a user frontruns a single `setConfig()` call to reach the cap before the config is changed, then `setConfig()` will reset all the buckets and the user can reach the cap again (second time) in a very short timespan, potentially in a single block.

Impact

Low. It is unlikely that this moment of opportunity would lead to a problem for TempleDAO, but the implementation of `setConfig()` does have a slight weakness.

Recommendation

The simple solution is to consider calling `setConfig()` from a flashbots relay. An alternative that would prevent even backrunning of `setConfig()` resetting the buckets is to redistribute the amounts from the existing buckets into the new bucket values. But this could be very gas intensive and add complexity to the circuit breaker design.

Developer Response

TempleDAO will execute via Flashbots protect when this function is required to be used. A comment has been added to the function to remind future devs of that fact.

5. Low - Weak liquidation incentives can result in unliquidated bad debt

The TLC has no incentive for the liquidator to liquidate an undercollateralized loan. There is no penalty applied to the lender for the liquidated position and the liquidator does not receive a percentage of the liquidation profits that would serve to pay for gas costs.

Technical Details

The liquidation of an undercollateralized position in TLC is performed by `batchLiquidate()`. The function checks if an address has an undercollateralized loan, and if so, the TEMPLE collateral is burned. No DAI is moved because the lender did not return the loaned funds, so only the collateral is in the protocol's possession. The burning of user-owned TEMPLE means that there is more treasury value per user-owned TEMPLE, increasing the protocol TPI. An increase in TPI indicates an increase in TEMPLE token price. This means that even though there is no liquidation incentive like there is in a protocol such as Compound Finance, TEMPLE holders should be incentivized to perform liquidations because it increases the value of the token they hold. The problem with this assumption is that it may fail in conditions with high gas prices or

where all token holders hold only small amounts of the token, leading to a bad debt situation that sits unresolved and grows worse due to accumulating interest.

Assume turbulent market conditions exist and cause mainnet gas prices to increase to $100 for a liquidation transaction. Further, assume that some loans might be eligible for liquidation due to interest accumulation, possibly from the same market turbulence.

If a user owns 1% of all circulating TEMPLE, then to make the $100 transaction profitable to execute, the liquidation would have to result in a $10,000 profit to the protocol. If the $10,000 profit represents the 15% overcollateralization of the loan, then the total loan size is $66,666, which is a very large loan.

Given this example, it may be no surprise that the protocol is designed in a way that the bot maintained by governance is expected to be the main liquidator for the protocol. This is because there is no incentive given to the liquidator except an increase in TPI (as long as the liquidation is profitable), and because the protocol multisig is the largest token holder at the current time, governance has the most incentive to liquidate positions and increase TPI. But in the case that governance is the primary liquidator, there is the risk that if governance liquidates positions when gas costs exceed liquidation profits, this impacts token value for all TEMPLE holders by lowering TPI because gas costs are paid by the same treasury that represents the value held by the TEMPLE token. There is the addition risk that if the bot goes offline, no one will have incentive to liquidate positions, but given the borrowing interest rate curve and active role of governance in this protocol, that is a much lesser risk.

This example shows that there is limited incentive for most TEMPLE token holders to liquidate undercollateralized positions. If the undercollateralized loan sits unresolved, the TEMPLE that serves as collateral to that loan is effectively weighing down the TPI. Even though it is not profitable for any user to liquidate the undercollateralized position, especially smaller loans, there will be no increase in TPI until gas prices drop enough to make the liquidation profitable (a quicker solution) or the loan accumulates enough interests to counteract the high gas prices (a slower solution).

The current design of TempleDAO is to use a governance-run bot to perform the liquidations, but this bot can encounter the same situation of unprofitable liquidations. If there are small value loans that are undercollateralized, the bot should not liquidate the loans when there are very high gas costs, because the protocol will lose value. It is also possible that the governance-run bot goes offline due to an outage in Google Cloud, AWS, or whichever server

provider(s) are used, so the on-chain bot cannot be relied on to have 100% uptime on par with a stable blockchain.

Impact

Low. Undercollateralized positions may not get liquidated quickly and in extreme circumstances may accrue bad debt.

Recommendation

There are different approaches that can reduce the risk of stale undercollateralized positions:

1. Giving the liquidator a small amount of the liquidated funds can increase the incentive to liquidate undercollateralized positions, but such an incentive can lead to toxic liquidation spirals as observed in Aave in November 2022.
2. A solution that does not need user interaction, and a solution that will be implemented according to conversations with the development team, is to implement a governance managed bot to perform liquidations. If this solution is implemented, it is important to document clearly how the bot operates. The bot should consider gas costs when performing liquidations to avoid losing value for the protocol with unprofitable liquidations.

Developer Response

TempleDAO will be running a liquidations bot (recommendation 2). A minimum user borrow limit has been added (`minBorrowAmount`), to mitigate small amounts of non-profitable bad debt (given gas spikes). Further, to optimise for gas consumption, multiple accounts can be liquidated in one call using `batchLiquidate()`

6. Low - Function selectors with a zero value are valid

Function selectors with a zero value are technically possible and should not be excluded due to validations.

Technical Details

Function selectors are represented by the first four bytes of the hash of the function signature. As these are the result of a hash function, it is possible and valid to have a function selector with the zero value (i.e. the first four bytes are zero).

There are two occurrences in the codebase in which function selectors taken as input arguments are validated to be different from zero:

- setExplicitAccess() in TempleElevatedAccess.sol

- setFunctionThresholdBatch() in ThresholdSafeGuard.sol

Impact

Low. All valid values should be permitted for the function selector.

Recommendation

Allow zero values and remove the validations.

Developer Response

Recommendation implemented and tests added in PR #842

7. Low - Aura rewards may be locked in AuraStaking

`getReward()` in AuraStaking.sol loops through the `rewardTokens` array to transfer reward tokens to the `rewardsRecipient` address. The `rewardTokens` array is only set in the constructor, so any new reward tokens that are added to Aura after this contract is deployed will not be possible to retrieve.

Technical Details

The `getReward()` function is intended to collect the reward tokens received from staking in Aura and send those tokens to `rewardsRecipient`. But the `rewardTokens` array is only set in the constructor and cannot be modified after. The Aura contract has a `addExtraReward()` function that can add to the existing list of reward tokens, but AuraStaking.sol is not designed for such flexibility. If AuraStaking.sol does not contain the full list of reward tokens, then the missing tokens will be sent from Aura to AuraStaking.sol and stay trapped in the TempleDAO contract.

Impact

Low. Rewards may be trapped depending on what rewards are distributed by Aura, but `recoverToken()` can be used for retrieval.

Recommendation

Create a setter function that allows the `rewardTokens` array to be updated. It can be similar to the setter function `setRewardsRecipient` that modifies the address that receives rewards.

Developer Response

Recommendation implemented and tests added in PR #842

8. Low - Borrowers may be instantly liquidated after rescue mode is disabled

The TempleLineOfCredit.sol contract can eventually enter a paused state if rescue mode is enabled. During this state, as debt is still accrued, a borrower's position may become unhealthy and be liquidated as soon as the pause is removed.

Technical Details

After rescue mode is enabled, all major functions of the contract are paused (`notInRescueMode` modifier). This means that repayments cannot be done by borrowers (since `repay()` and `repayAll()` are blocked), but their debt is still accrued.

If a position becomes unhealthy while the rescue mode is enabled, a borrower can be liquidated as soon as the pause is disabled, and before they can even attempt to normalize their situation.

Impact

Low. Rescue mode should be a rare event.

Recommendation

After rescue mode is disabled, allow a grace period before enabling liquidations back. This can be implemented by a toggle (e.g. `liquidationsPaused`) that can be enabled just before rescue mode is disabled.

Developer Response

Reccomendation implemented with tests in PR #842

9. Low - Token allowance isn't revoked when updating TRV in Gnosis strategy

Both `repay()` and `repayAll()` set an infinite token allowance to the current treasury vault that isn't revoked if the vault contract is updated.

Technical Details

When repayments to the treasury vault are made from the Gnosis strategy, both `repay()` and `repayAll()` will execute an infinite (i.e. `type(uint256).max`) token approval so that the vault can pull the tokens from the strategy contract.

The treasury vault contract can be updated in strategies, however, the implementation of `_updateTrvApprovals()` is empty for the GnosisStrategy.sol contract. This means that previously configured vaults will still have unrestricted access to tokens held by the strategy contract.

Impact

Low. Unnecessary token allowances extend trust boundaries unnecessarily and increase the risk of loss of value.

Recommendation

Approve a finite amount based on the repayment amount or track which tokens have been granted an infinite approval, so that those approvals can be revoked (i.e. set to zero) when the vault is updated (in `_updateTrvApprovals()`).

Developer Response

Recommendation implemented and tests added in PR #842

10. Low - Validate rate parameters in LinearWithKinkInterestRateModel.sol

The kinked interest rate model is defined by three different configuration parameters: `baseInterestRate`, `kinkInterestRate` and `maxInterestRate`. `baseInterestRate` is the rate when the utilization ratio is 0, `kinkInterestRate` is the rate at the kink and `maxInterestRate` is the rate at 100% utilization. This means that `kinkInterestRate` should not be less than `baseInterestRate`, and that `maxInterestRate` should not be less than `kinkInterestRate`.

Technical Details

When the utilization ratio is above the kink, the slope of the linear model is defined as `(maxInterestRate - kinkInterestRate) / kinkUtilizationRatio`. This means that `maxInterestRate >= kinkInterestRate`.

Similarly, in the other case, the slope is defined as `(kinkInterestRate - baseInterestRate) / kinkUtilizationRatio`, which implies that `kinkInterestRate >= baseInterestRate`.

Impact

Low. The calculation will overflow and revert if the order is not followed.

Recommendation

Validate that `baseInterestRate <= kinkInterestRate <= maxInterestRate` in `_setRateParams()`.

Developer Response

Recommendation implemented and tests added in PR #842

11. Low - Account validation in TempleElevatedAccess.sol

Both `rescuer` and `executor` accounts play a key role in contract administration and should be validated to be correctly initialized.

Technical Details

The constructor of TempleElevatedAccess.sol initializes `initialRescuer` and `initialExecutor` without checking for zero/empty addresses.

Additionally, as the `rescuer` account is intended to be a support role of the `executor` account, it is important to validate that both accounts are not the same address. The design intent is for `rescuer` and `executor` to be multisigs and not EOA, so an extra check can be added to verify these addresses are contracts.

Impact

Low. Improper or loose validation can break assumptions about how the protocol should operate.

Recommendation

- In the constructor, check that `initialRescuer != address(0)`, `initialExecutor != address(0)` and `initialRescuer != initialExecutor`.
- In `acceptRescuer()`, validate that the new `rescuer` is not the current `executor`, i.e. `require(msg.sender != executor)`.
- In `acceptExecutor()`, validate that the new `executor` is not the current `rescuer`, i.e. `require(msg.sender != rescuer)`.
- In `proposeNewRescuer()` and `proposeNewExecutor()`, verify that `account.code.length > 0`. This can replace the existing check of `account != address(0)`. The same check should be added to the constructor.

Developer Response

Recommendations 1-3 implemented and tests added in PR #842

The fourth recommendation has not been implemented, because it should not be an assumption that the rescuer/executor is always a smart contract. For example if Fireblocks were to be used in the future, this will be an EOA (using MPC)

12. Low - Zero token transfers while claiming rewards in AuraStaking.sol

The `getReward()` function doesn't check for potential zero token transfers, which may lead to an accidental denial of service while claiming rewards in AuraStaking.sol.

Technical Details

Some ERC20 implementations revert on zero value transfers. The implementation of the `getReward()` function in the AuraStaking.sol contract executes a token transfer of the

contract's token balance without actually checking if the balance is not zero.

Since all reward tokens are pulled and transferred simultaneously, a failure in any of the token transfers would result in a transaction revert, causing a denial of service in the reward claiming process.

Impact

Low. Missing zero token validation could cause problems for certain ERC20 tokens.

Recommendation

Check if the amount is greater than zero before executing the transfer.

```
function getReward(bool claimExtras) external override {
    IAuraBaseRewardPool(auraPoolInfo.rewards).getReward(address(this), claimExtras);
    if (rewardsRecipient != address(0)) {
        uint256 length = rewardTokens.length;
        for (uint i; i < length; ++i) {
            uint256 balance = IERC20(rewardTokens[i]).balanceOf(address(this));
-            IERC20(rewardTokens[i]).safeTransfer(rewardsRecipient, balance);
+
+            if (balance > 0) {
+              IERC20(rewardTokens[i]).safeTransfer(rewardsRecipient, balance);
+            }
        }
    }
}
```

Developer Response

Recommendation implemented in PR #842

13. Low - Insufficient gas for max buckets in TempleCircuitBreakerAllUsersPerPeriod.sol

TempleCircuitBreakerAllUsersPerPeriod.sol sets a `buckets` array size of 65535. But an insufficient gas revert will happen if the constructor is passed a value as small as 4500 for `_nBuckets`.

Technical Details

TempleCircuitBreakerAllUsersPerPeriod.sol comments and the declaration of `buckets` indicate that the design is intended to allow up to 65535 buckets. But it is not easy to create anywhere near this number of buckets directly, because `setConfig()` reverts due to lack of gas if less than 10% that many buckets are supplied in the constructor. This is because `_setConfig()` clear the bucket values by looping through all buckets to set their initial value to 1. Now because of how solidity memory works, it may be possible for governance to clear 4000 buckets of array space at a time without running out of gas. But then in order to reach 65k buckets, `setConfig()` would need to be called over 10 times, incrementally increasing the value of `nBuckets` each time. Overall, the current design means that realistically the maximum number of buckets is too large, and the `buckets` array is oversized for an ETH mainnet deployment with a 30 million gas limit.

Impact

Low. Comments and design cannot realistically be used to the maximum extent intended without reverting.

Recommendation

Consider reducing the maximum number of buckets to a more reasonable number given gas constraints, such as 4000.

Developer Response

Recommendation implemented and tests updated in PR #842

14. Low - `addStrategy()` allows setting debt ceiling for tokens without borrowing enabled

`setStrategyDebtCeiling()` and `addStrategy()` have similar logic to set the debt ceiling of a strategy, but `setStrategyDebtCeiling()` has a safety check that `addStrategy()` is missing.

Technical Details

`setStrategyDebtCeiling()` checks if a token is a borrowable token before setting the debt ceiling of that token with the line `if (!_borrowTokenSet.contains(address(token))) revert BorrowTokenNotEnabled();`. When the debt ceiling is set in `addStrategy()`, there is no such check to confirm that the token is a borrowable token. This means the two functions have different checks even though they have the same `onlyElevatedAccess` modifier controlling who can call these functions.

Impact

Low. The same action has different protective measures in place in different locations.

Recommendation

Modify `addStrategy()` to check if a token is borrowable before setting the debt ceiling, similar to the logic in `setStrategyDebtCeiling()`.

Developer Response

Recommendation implemented and tests updated in PR #842

15. Low - `shutdown()` doesn't delete `strategyTokenCredits` values

`shutdown()` deletes a strategy and values related to that strategy. The `strategyTokenCredits` values associated with the strategy are not deleted, but should be.

Technical Details

The goal of `shutdown()` in TreasuryReservesVault.sol is to remove all data related to a strategy from variables and mappings. Although the credit value stored in the `strategyTokenCredits` mapping is used in the `StrategyShutdownCreditAndDebt` event, the value is never deleted.

Impact

Low. In order to fully remove the strategy from the vault, all associated values should be deleted.

Recommendation

```
    for (uint256 i; i < _length; ++i) {
        _token = _borrowTokenSet.at(i);
        _outstandingDebt = borrowTokens[IERC20(_token)].dToken.burnAll(strategy);
        emit StrategyShutdownCreditAndDebt({
            strategy: strategy,
            token: _token,
            outstandingCredit: credits[IERC20(_token)],
            outstandingDebt: _outstandingDebt
        });

        // Clean up the debtCeiling approvals for this borrow token.
        // Old borrow ceilings may not be removed, but not an issue
        delete _strategyConfig.debtCeiling[IERC20(_token)];
+       delete credits[IERC20(_token)];
    }
```

Developer Response

Recommendation implemented and tests updated in PR #842

16. Low - Casting overflow risk in strategy borrowing

Values cast from uint256 to int256 may overflow. SafeMath does not apply to casting, so the overflow would result in a negative value.

Technical Details

`_availableForStrategyToBorrow()` calculates how many more tokens a strategy is allowed to borrow. To prevent a revert due to underflow, the function casts uint256 values to int256 values. Casting has no overflow protection, and because the `debtCeiling` value can be set to any arbitrary integer, it is possible that a very high ceiling value would result in an overflow. An overflow of the sum `_ceiling + _credit` would not give a strategy more borrowing than it should, but an overflow of `dTokenBalance` could cause `int256(dTokenBalance)` to resolve to a negative value, increasing the strategies borrowing greater than it should be. dTokens are minted 1:1 for the amount of tokens borrowed, so it is unlikely to reach a value that is large enough to overflow the int256 data type, but a modified implementation for this function could remove the problem entirely.

Impact

Low. Low probability overflow would increase a strategy's maximum borrowing amount.

Recommendation

Consider a different implementation for this math logic. Instead of casting to int256, use unchecked math and at each stage check whether the result overflows the data type or not.

Developer Response

Fixed and tests added in PR #842

Instead of checking for overflow within `_availableForStrategyToBorrow()` (in a view after the offending transaction which would make this overflow), we are now preventing the overflow to occur. Whenever a borrow, repay or debt ceiling change occurs the potential overflow with `_ceiling + _credit` is checked and reverts on overflow.

17. Low - Ramos inconsistencies between constructor and setter functions

Ramos.sol limits the values that can be set for some state variables in the setter functions. But these limits are not applied in the contract constructor.

Technical Details

1. `setPostRebalanceSlippage()` reverts when trying to set `postRebalanceSlippage` as zero. Because `postRebalanceSlippage` is zero by default, and it is not set in the constructor, it will remain zero until `setPostRebalanceSlippage()` is called. The same issue of a default zero value until otherwise set applies to `setMaxRebalanceAmounts()`, `setRebalancePercentageBounds()`, `setTpiOracle()`, `setTokenVault()`, and `setRebalanceFees()`. `feeCollector()` does not have this issue because `feeCollector` is set in the constructor.
2. The constructor does not check if any state variables are set to valid values, but the setter functions do introduce this check. This means that the value set in the constructor may not align with the assumptions made elsewhere in the contract. For example, `feeCollector` cannot be set to the zero address value in `setFeeCollector()`, but can be set to the zero address value in the constructor. The `maxRebalanceFee` value implicitly should be less than `BPS_PRECISION` due to the accounting logic in the contract, but there is no logic in the constructor to confirm that `maxRebalanceFee < BPS_PRECISION`.

Impact

Low. Different logic applies to key variables in different places.

Recommendation

1. Set all necessary values for the contract to function in the constructor. This avoids calling many setter functions after the contract is deployed.
2. Add zero checks in the constructor for relevant values

Developer Response

It is not possible to add more values to the constructor, as `stack too deep` issues arise. The values passed through to the constructor are all immutable (so need to be set in the constructor), and `feeCollector` which also needs to be set at constructor since the `setFeeCollector()` can only be called by the existing `feeCollector`

Further, the constructor is called from a source controlled deployment script which runs the post deployment steps (eg calling `setPostRebalanceSlippage()`). Deployed contracts are thoroughly verified post deployment.

The only extra verification that was made was to the `_maxRebalanceFee`. Implemented and tests added in PR #842

18. Low - Hardcoded Maker DSR base rate doesn't match on-chain value

TempleDebtToken.sol hard-codes a `baseRate` value that determines the compounding base interest rate for the token. There are three issues with this in the case where the underlying DAI stablecoins are deposited into the MakerDAO DSR:

1. The deployment script has a value of 3.49% for the Maker DSR, which is the old rate before it was changed in block 17722372 (July 18, 2023).
2. MakerDAO's DSR value is stored as an interest rate per second with 1e27 decimals of precision. But the `baseRate` value in TempleDebtToken.sol is described as `The interest rate per annum` in a comment in CompoundedInterest.sol. The difference in precision and how the interest is calculated is likely to lead to rounding differences that may increase over time, causing the interest accumulation between TempleDAO and MakerDAO's DSR to differ.
3. The `baseRate` is set by TempleDAO governance, but Maker's DSR rate can also be set by MakerDAO's governance. Unless TempleDAO governance updates the DSR rate in the exact same block that MakerDAO updates their DSR rate, the accumulated interest calculated by TempleDAO and MakerDAO will not match.

Technical Details

There are two issues related to how TempleDAO handles the MarkerDAO DSR rate:

1. The first issue is easy to fix. The deployment script uses the old MakerDAO DSR value of 3.49%, but the new value is 3.19%. This is confirmed by makerburn.com and by querying the on-chain value:

```
cast call 0x197E90f9FAD81970bA7976f33CbD77088E5D7cf7 "dsr()(uint256)" --block 17722371
```

Return value: 1000000001087798189708544327 (equates to 3.49%)

```
cast call 0x197E90f9FAD81970bA7976f33CbD77088E5D7cf7 "dsr()(uint256)" --block 17722372
```

Return value: 1000000000995743377573746041 (equates to 3.19%)

1. The second and third issues suggest that a redesign may be a better approach to properly handle integration with MakerDAO's DSR. Instead of storing a `baseRate` value and calculating the accumulated interest, these operations should be outsourced to MakerDAO's contracts to avoid conflicting accounting calculations. If mismatched

accounting is allowed to operate over long timespans, the delta between the actual value accrued from MakerDAO's DSR and the estimated interest accumulated calculated by TempleDAO can grow. If the value is overestimated, this can lead to bad debt and users can be misinformed about the interest rate they expect they are getting with the MakerDAO DSR integration.

Although the ideal case would be for the base rate to match the Maker DSR rate, the developer intent is to allow the base rate to change in the future to be based on a different protocol. The contract therefore cannot be designed only around MakerDAO. Additionally, governance has options to adjust the accumulated interest (by minting/burning debt tokens or temporarily overcorrecting the base rate for a short period) if precision of the internal credit and debt becomes important later. For now, the internal credit and debt is only useful for governance to compare the performance of strategies against one another, and because the comparison is relative, precise values matching the MakerDAO DSR rate will not impact the comparison process.

Impact

Low. Actual interest accumulated by Maker DSR will likely differ from TempleDAO's interest calculations. This can cause a mismatch of estimated value and actual value, but governance only cares about the relative performance of strategies, so precise matching of the Maker DSR rate is not crucial.

Recommendation

The DSR value and associated interest accumulation calculations should not be handled by TempleDAO contracts, but instead should rely on the external MakerDAO contracts' calculations.

Developer Response

The dUSD interest rate serves as a fully internal baseline rate for Strategy performance relative to the lowest risk Strategy available to the Treasury, also known as the risk-free rate (RFR). We agree that an overly stale benchmark rate for dUSD is not desirable. However, it does not follow that we should directly pull the DSR rate from the Maker contract.

Every TRV integration to an external protocol adds risk. Given that the base rate interacts with ALL Strategy borrowers, it would be an extremely high risk integration. Any exploit or mishap on the Maker side could wreck havoc because the RFR is a critical parameter that interacts with all Strategies. If we switch to another Strategy as our lowest risk strategy in the future, we would have to re-integrate with a new Protocol just to get the right RFR number.

To mitigate staleness without hardcoded integration to Maker, we can set up a bot to monitor and send Alerts to Discord and update the benchmark rate on our V2 dashboard on a regular basis.

Gas Saving Findings

1. Gas - Simplify logic in `removeLiquidity()` function

In `removeLiquidity()`, the repayment logic can be simplified to remove the iteration and save gas.

Technical Details

In the implementation of `removeLiquidity()`, tokens are repaid to the strategy by iterating the array of assets and checking if each asset corresponds to the quote or protocol token. This is not really needed and can be simplified by removing the loop and just repaying both tokens.

Impact

Gas savings.

Recommendation

Repay both protocol and quote token directly without using a loop.

```
...
unchecked {
    protocolTokenAmount = protocolToken.balanceOf(address(this)) -
protocolTokenAmountBefore;
    quoteTokenAmount = quoteToken.balanceOf(address(this)) - quoteTokenAmountBefore;
}

if (protocolTokenAmount != 0) {
    tokenVault.repayProtocolToken(protocolTokenAmount);
}

if (quoteTokenAmount != 0) {
    tokenVault.repayQuoteToken(quoteTokenAmount);
}
...
```

Developer Response

Recommendation implemented and tests updated in PR #842

2. Gas - Avoid burn logic if amount is zero

In TempleDebtToken.sol, the `burn()` function will execute its logic even if the amount to burn is zero.

Technical Details

The amount of tokens to burn is limited by the actual debt amount of the debtor. This means that, even if the original argument (`_burnAmount`) is greater than zero, it can be overridden to zero if the debtor doesn't have any debt. If this is the case, the implementation will still call the internal function `_burn()`, wasting gas in operations that won't alter the state.

This is particularly relevant, in the context of the vault, for strategies that execute repayments without having any actual debt. Repayment logic calls the `burn()` function of the debt token to know how much excess should be accounted for as credit.

Impact

Gas savings.

Recommendation

Check if the capped `_burnAmount` is greater than zero to avoid calling `_burn()` if it is not needed.

```
    function burn(
        address _debtor,
        uint256 _burnAmount
    ) external override returns (
        uint256 burnedAmount
    ) {
        if (!minters[msg.sender]) revert CannotMintOrBurn(msg.sender);
        if (_debtor == address(0)) revert CommonEventsAndErrors.InvalidAddress();
        if (_burnAmount == 0) revert CommonEventsAndErrors.ExpectedNonZero();

        Debtor storage debtor = debtors[_debtor];
        uint256 _totalPrincipalAndBase = _compoundedBaseInterest();
```

```
        {
            // The user can't pay off more debt than they have.
            // It is capped, and the actual amount burned returned as a value
            uint256 _debtorBalance = _balanceOf(debtor, _totalPrincipalAndBase);
            if (_burnAmount > _debtorBalance) {
                _burnAmount = _debtorBalance;
            }
        }

+       if (_burnAmount > 0) {
            emit Transfer(_debtor, address(0), _burnAmount);
            _burn(debtor, _burnAmount, _totalPrincipalAndBase);
+       }

        return _burnAmount;
    }
```

Developer Response

Recommendation implemented and tests updated in PR #842

3. Gas - Duplicate and unneeded logic in `repayAll()`

The `repayAll()` function shares some common logic with `repay()` through the internal function `_repayToken()`, though this function includes some functionality that is not needed in the case of `repayAll()`.

Technical Details

Shared logic present in the `_repayToken()` contains code which is not really needed for the case of `repayAll()`.

- Account debt is recalculated (calls to `_currentAccountDebt()`).
- The check to validate that the amount does not exceed the debt is not needed (since `_repayAmount` equals `_newDebt`).
- `debtCheckpoint` can be set directly to zero, as it is paying the full debt.

Impact

Gas savings.

Recommendation

Refactor `_repayToken()` to avoid unneeded logic for the case of `repayAll()`.

Developer Response

Recommendation implemented in PR #842

4. Gas - Move debt token cache out of the loop in `computeLiquidity()`

The debt token cache is initialized in each iteration of the loop present in the `computeLiquidity()` function.

Technical Details

In `computeLiquidity()`, the debt token cache struct is data independent in the for loop and can be moved outside to improve gas costs.

Impact

Gas savings.

Recommendation

Initialize `_debtTokenCacheRO()` outside the for loop.

Developer Response

Recommendation implemented in PR #842

5. Gas - Debt token cache is re-initialized in `borrow()` function

The debt token cache struct is initialized twice in the `borrow()` function.

Technical Details

The `borrow()` function first initializes the cache in line 204 and then calls the `_checkLiquidity()` internal function which also initializes the same cache.

Impact

Gas savings.

Recommendation

Refactor `_checkLiquidity()` so that it can receive the cache variable as a parameter to avoid the re-initialization.

Developer Response

Recommendation implemented in PR #842

6. Gas - Change visibility of public constants

Public constants automatically generate getter functions which increase code size and deployment costs.

Technical Details

Several contracts define a `VERSION` constant that is marked as public, while at the same time expose the same constant value in an explicit getter function.

Examples:

- TempleDebtToken.sol
- TreasuryReservesVault.sol
- DsrBaseStrategy.sol

Impact

Gas savings.

Recommendation

Change the visibility of the constants to `private`.

Developer Response

Recommendation implemented in PR #842

7. Gas - Cache storage variable locally to prevent multiple reads from storage

Cache variables read from storage to prevent multiple SLOAD operations.

Technical Details

- `rewardsRecipient` in `AuraStaking::getReward()`
- `rewardTokens[i]` in `AuraStaking::getReward()`
- `poolHelper` in `Ramos::rebalanceUpExit()`
- `poolHelper` in `Ramos::rebalanceDownExit()`
- `poolHelper` in `Ramos::rebalanceUpJoin()`
- `poolHelper` in `Ramos::rebalanceDownJoin()`
- `tokenVault` in `Ramos::addLiquidity()`
- `tokenVault` in `Ramos::removeLiquidity()`
- `totalPrincipal` in `TempleDebtToken::currentTotalDebt()`

- `maxTreasuryPriceIndexDelta` in `TreasuryPriceIndexOracle::setTreasuryPriceIndex()`
- `rescuer` in `TempleElevatedAccess::proposeNewRescuer()`
- `cap` in `TempleCircuitBreakerAllUsersPerPeriod::preCheck()`
- `treasuryReservesVault` in `DsrBaseStrategy::trvWithdraw()`
- `treasuryReservesVault` in `GnosisStrategy::borrowMax()`
- `treasuryReservesVault` in `GnosisStrategy::repay()`
- `treasuryReservesVault` in `GnosisStrategy::repayAll()`
- `treasuryReservesVault` in `TempleTokenBaseStrategy::trvWithdraw()`
- `_accountData.collateral` in `TempleLineOfCredit::removeCollateral()`

Impact

Gas savings.

Recommendation

Cache state in local variables instead of reading again from storage.

Developer Response

All recommendations implemented in PR #842, except:

- `maxTreasuryPriceIndexDelta` in `TreasuryPriceIndexOracle::setTreasuryPriceIndex()`
- `cap` in `TempleCircuitBreakerAllUsersPerPeriod::preCheck()`

These two don't make sense to implement - we don't want to use extra gas on the happy path (by creating a new variable). If it's going to revert, paying the tiny amount of extra gas is fine

8. Gas - Unneeded unsafe downcasting in debt token cache

The `interestAccumulator` field in the `DebtTokenData` struct is of type `uint256` and doesn't need to be downcasted to `int128` while building the cache.

Technical Details

The `DebtTokenCache` struct is used in TempleLineOfCredit.sol as a cache to initialize several values from storage to be used in different functions of the contract.

In this cache struct, the `interestAccumulator` field is defined as `uint128` while its matching field in the struct used in storage is defined as `uint256`, leading to an unsafe downcast while initializing the struct in `_initDebtTokenCache()`

Impact

Gas savings.

Recommendation

Change `interestAccumulator` to `uint256` in `DebtTokenCache` to match the data type of `DebtTokenData`.

Developer Response

Recommendation implemented in PR #842

9. Gas - Set proper types to remove SafeCast operations

SafeCast is used in TreasuryPriceIndexOracle.sol to cast function argument values. But if the function arguments are the proper type instead of uint256, there is no need for any casting, saving gas in all the setter functions, and this import can be removed to save gas on deployment.

Technical Details

Multiple setter functions in TreasuryPriceIndexOracle.sol cast function arguments to smaller integer types (1, 2, 3). This casting is unnecessary if the function argument is the proper type.

Impact

Gas savings.

Recommendation

Set function arguments to the proper type to avoid casting. For example, `_initialTreasuryPriceIndex` should be uint96, `cooldownSecs` should be uint32, and `value` should be uint96. Finally, remove the SafeCast import when it is not needed.

Similar improvements can be made to function arguments in other contracts relying on SafeCast, including LinearWithKinkInterestRateModel.sol, TempleDebtToken.sol, TempleLineOfCredit.sol, and TempleCircuitBreakerAllUsersPerPeriod.sol.

Developer Response

Recommendation implemented and tests updated in PR #842

10. Gas - Use unchecked if no underflow risk

There are subtraction operations that can use unchecked for gas savings.

Technical Details

The subtractions in `_mintDToken()` can be unchecked because they are inside if/else branches with logic that prevents an underflow from happening (L569, L577).

The calculation of `_remaining` in `_burnDToken()` can be made unchecked because `_burnedAmount` is the same as the value of `toBurnAmount` or less, based on the logic found in dToken `burn()`.

The calculation of `_delta` in `setTreasuryPriceIndex()` can be unchecked.

The calculation of `_newDebt` in `_repayTotalDebt()` can be unchecked.

In `_burn()`, the updates to the `_burnAmount` variable can be unchecked, as the repaid amount should be less than the amount to burn (L275, L284, L293, L297).

Impact

Gas savings.

Recommendation

Use unchecked block if there is no overflow or underflow risk for gas savings. Consider adding a fuzzing test to verify the underflow cannot happen to add a safety check for future code modifications.

Developer Response

Recommendation implemented in PR #842

11. Gas - `enoughCooldown` modifier can be simplified

The `lastRebalanceTimeSecs != 0` check in the `enoughCooldown` modifier can be removed.

Technical Details

`enoughCooldown` performs two checks:

```
1 lastRebalanceTimeSecs != 0
2 lastRebalanceTimeSecs + cooldownSecs > block.timestamp
```

In the case that `lastRebalanceTimeSecs` is zero, the 2nd condition will return false assuming `cooldownSecs` is set to a value less than `block.timestamp`. This means the first condition can be removed from the if statement to save gas.

Impact

Gas savings.

Recommendation

Remove the `lastRebalanceTimeSecs != 0` check from the `enoughCooldown` modifier.

Developer Response

Recommendation implemented in PR #842

Informational Findings

1. Informational - Upgrade outdated dependencies

The gnosis safe npm package used as a dependency, @gnosis.pm/safe-contracts, has not been updated in 2 years despite newer versions available from the main github repository.

The prb/math dependency has a newer version out that may be best to upgrade to.

Technical Details

`package.json` lists @gnosis.pm/safe-contracts version 1.3.0 as a dependency. This version is the latest version available in the npm package but v1.4.1 is the latest available from the main gnosis repository.

While the dependency is not used extensively though and is only imported into ThresholdSafeGuard.sol, SafeForked.sol, and IThresholdSafeGuard.sol, dependencies should remain updated to include the latest bug fixes.

The prb/math dependency might be an easier to change because the npm package is update. Only the version number needs upgrading from 3.3.1 to the latest 4.0.1.

Impact

Informational.

Recommendation

Use the github repo code instead of the npm package. The gnosis team should update their own npm package.

Also consider updating the prb/math package to the latest release.

Developer Response

The prb/math dependency has been updated to `4.0.1`, this also required updating the solc version to `0.8.19`. Updated in PR #842

Gnosis Safe 1.3.0 will not be upgraded because the Gnosis UI and deployed Safe's are still using v1.3.0. For contract compatibility the Safe Threshold should be pinned to the same version.

2. Informational - Incorrect variable type in BalancerPoolHelper

BalancerPoolHelper.sol has two similar functions `createPoolJoinRequest()` and `createPoolExitRequest()`. The return value of these functions is a struct, but there is an inconsistency in the interface defining these structs which does not match Balancer's code.

Technical Details

The two return value structs are defined in IBalancerVault.sol. Observe that `assets` is of type IERC20 in `JoinPoolRequest` but of type `address` in `ExitPoolRequest`.

```
struct JoinPoolRequest {
  IERC20[] assets;
  uint256[] maxAmountsIn;
  bytes userData;
  bool fromInternalBalance;
}

struct ExitPoolRequest {
  address[] assets;
  uint256[] minAmountsOut;
  bytes userData;
  bool toInternalBalance;
}
```

In comparison, these structs should have the same types according to the Balancer docs.


types

The on-chain Balancer implementations of `JoinPoolRequest` and ExitPoolRequest also shows consistency in variable types between the structs, although type `IAsset` is used instead of type `address`.

Impact

Informational.

Recommendation

The `assets` array should have type `address` in the JoinPoolRequest struct. Update the type for array `assets` in `BalancerPoolHelper.createPoolJoinRequest()`.

Developer Response

Recommendation implemented in PR #842

3. Informational - High centralization risk throughout protocol

The TempleDAO contracts are architected in a way that involves a high level of centralization risk. Centralization can offer many benefits like faster response times from governance to adapt the protocol, but also downsides. Users interacting with the protocol should be aware of the risks involved with this design choice because rogue governance can withdraw protocol value to arbitrary addresses.

Technical Details

Every DeFi protocol must decide how immutable or centralized the protocol will be. TempleDAO's design falls on the very centralized end of the spectrum. This by design and does not add risks if governance actions are only performed by trusted parties. But because this assumption is not governed by immutable code, some users may not be convinced that this assumption holds at all time. If this assumption does not hold, governance can:

1. Withdraw tokens directly from the protocol contracts including `AuraStaking.recoverToken()`, `AuraStaking.withdrawAllAndUnwrap()`, `AuraStaking.withdrawAndUnwrap()`, `Ramos.recoverToken()`, `TempleLineOfCredit.recoverToken()`, `TreasuryReservesVault.recoverToken()`, etc.
2. The majority of public or external functions in TempleDAO v2, especially in RAMOS and the strategies, has access control modifiers that allow only a governance multisig to call these functions. Many functions control key state variables that influence fees,
3. Off-chain bots controlled by governance play a key role in performing swaps to stabilize the TEMPLE token price and handle liquidations. It was not clear during this audit what the off-chain bot implementation looks like and what the security around its operation looks like.

In summary, the centralized nature of TempleDAO adds some risk because of the power that governance has. This finding has a risk of Informational only because the assumption was made that the protocol has trustworthy governance. If this assumption does not hold, the risk severity would likely jump to Critical. Even if all governance operates as a trusted party,

centralization brings with it the risk of private key theft that has impacted several DAOs in the last couple of years.

Impact
Informational.

Recommendation
The centralized design choices have likely been made for good reason, but it would help to document very clearly what safety measures are taken around the multisig and governance models that will be involved in v2 to minimize the risk of malicious actions (intentional or not). Setting different signature criteria for different functions, depending on the possible impact of each function, is one idea mentioned by the developers, but that was out of scope of this review.

Developer Response
The fundamental assumption of V2 is that elections will produce or re-affirm trusted multisig signers who will not collude against the DAO to act in their own self-interest. In our two year history, a small group of trusted multisig SAFE owners have signed transactions on behalf of the DAO. In the past when the Treasury has suffered a loss of access to funds and could no longer ascertain the TPI, or had outsized exposure to a depegging event, this same group of signers was empowered to act in a prompt and decisive manner to keep remaining Treasury funds safe.

While we could make the execution of Treasury transactions more decentralized, it is unclear what we would gain from a security perspective by moving away from a proven system that has worked. Retrospectively, we can surmise that a more decentralized system would have failed to protect user funds from actual events that did transpire. To accept a known downgrade in terms of user fund security in exchange for unknown but probably minor benefits at the multisig execution level is not appealing to us.

Instead, we aim to relinquish decentralization elements to the political process and the signature authority process while ensuring that those elected signers are imbued with the centralized operator power they need to best serve the needs of the DAO at large.

4. Informational - Add events to RamosStrategy.sol

RamosStrategy.sol lacks events in borrow and repay functions. Other strategies have events in corresponding functions.

Technical Details

The borrow and repay functions in RamosStrategy.sol do not emit events. This is unlike the other events such as DsrBaseStrategy.sol and TempleTokenBaseStrategy.sol.

Impact

Informational.

Recommendation

Add events to RamosStrategy.sol borrow and repay functions, as well as any other critical functions that handle value transfer for easier on-chain analysis and debugging in edge case scenarios.

Developer Response

Recommendation implemented and tests updated in PR #842

5. Informational - TLC lending curve configuration choices may result in low utilization ratio

The ability to borrow DAI from TempleLineOfCredit.sol is similar to other protocols that allow borrowing on deposited collateral. The values chosen for the lending curve in the deployment script may not be ideal for encouraging lending.

Technical Details

The values set in the deployment script for LinearWithKinkInterestRateModel.sol result in a significantly different borrowing curve compared to Compound v2 and Aave DAI markets. Consider the goals and risk implications of the current choices to determine how closely TempleDAO aims to align with these other protocols to remain competitive and maintain a utilization ratio in line with the DAO's goals. If the lowest borrowing rate at zero utilization is 5%, this may be too high of a borrowing rate with current market conditions to remain competitive. Compound and Aave regularly modify the values that determine their borrowing curves based on feedback from Gauntlet that accounts for current market conditions.

The suggestion to lower the interest rate at zero utilization is only valid with the current Maker DSR rate of 3.19%. If Maker increases the DSR rate, such as with the recent eDSR proposal that passed, then a different y-intercept value would make sense nearer to whatever base rate DAI in the TempleDAO base strategy is earning.

Impact

Informational.

Recommendation

Consider altering the borrowing curve configuration values such that:

1. The y-intercept is the same as the Maker DSR rate (currently 3.19%), possibly calling the external `pot.dsr()` like DsrBaseStrategy.sol.
2. The borrow rate at the location of the kink should be less than 10% to align more closely to Compound and Aave, most likely closer to 5-7%.

The interest curve choice is just a design decision made by protocol governance. The values observed in the deployment script indicate there is no obvious security risk in normal market conditions with the current values, perhaps just minor optimizations that can be made. Improved modeling using current market conditions and the goals of lending protocol can further inform the values chosen.

Developer Response

The interest rate policy in TLOC is NOT primarily intended to force repayment + encourage more Supply side liquidity when UR% is high as might be the case for Aave or Compound. TThe cost of borrowing in the TLOC should be viewed as an expected rate of return for TLOC and should be compared to the rate of return achievable from other V2 Strategies because TLOC itself is a V2 Strategy.

If the Treasury is completely AGNOSTIC with respect to whether it deploys its DAI in TLOC or DSR, the minimum borrow interest rate in TLOC should be 8% and the rate at the kink should reflect some premium over the risk-free rate e.g. 12% to price in the cost of illiquidity i.e. TLOC loans are not immediately available to the Treasury unlike funds in the DSR. The maximum interest rate could be 20% representing the highest yielding non-TLOC V2 strategy.

Through its interest rate curve policy, the Treasury wants to ensure that it can close any potential user-driven positive spread, or carry trade that involves borrowing from TLOC and investing in another Strategy that the Treasury could also deploy to if it had the desire to do so. We expect to see an equilibrium reached between the TLOC borrow rate and whatever APY that the alternative Strategy is able to achieve as the UR% for TLOC goes up or down (risk-adjusted).

6. Informational - Potential denial of service in TempleLineOfCredit.sol contract

A bad actor can artificially exhaust the limits of the circuit breaker, causing a temporary denial of service in the contract's functionality.

Technical Details

The TempleLineOfCredit.sol contract contains a circuit breaker safety measure in the `removeCollateral()` and `borrow()` functions. The current implementation of the circuit breaker (TempleCircuitBreakerAllUsersPerPeriod.sol) enforces a global volume limit over a sliding window period of time.

As these limits apply globally to all users, any account can artificially generate volume to hit the caps defined by the circuit breaker, causing a temporary denial of service in the contract.

For example, a malicious user can repeatedly call the `addCollateral()` and `removeCollateral()` functions to exhaust the limits of the circuit breaker and make the `preCheck()` call revert for all other users.

Impact

Informational.

Recommendation

The issue reveals a design trade-off with the current circuit breaker solution: even if there is no intentional attack, the limits can be reached by users with a large volume of tokens (e.g. another protocol). Since caps can be dynamically adjusted by governance, the protocol should actively monitor these interactions off-chain to be alerted in case such a scenario occurs.

Developer Response

Protecting user funds is paramount for us which is why we implemented the circuit breaker. The utilisation and limits will be actively monitored for abuse – we expect an iterative approach to the circuit breaker implementation. It was designed in a modular and flexible way such that we have options to amend the implementation of this going forward if required.

7. Informational - Unused fields in `AccountData` struct

The `AccountData` struct present in the ITlcDataTypes.sol interface has unused fields that can be safely removed.

Technical Details

The following fields from the `AccountData` struct are not used throughout the codebase:

- `removeCollateralRequestAmount`

- `borrowRequestAmount`
- `removeCollateralRequestAt`
- `borrowRequestAt`

Impact

Informational.

Recommendation

Remove the unused fields from the struct.

Developer Response

Recommendation implemented in PR #842

8. Informational - Replace magic numbers with constants

Constant variables should be used in place of magic numbers to prevent typos. For one example, the magic number `65535` is found in TempleCircuitBreakerAllUsersPerPeriod.sol and can be replaced. Using a constant or other variable also adds a description to the value to explain the purpose of the value.

Technical Details

The value `65535` appears in an if statement in TempleCircuitBreakerAllUsersPerPeriod.sol, but the value represents the length of the `buckets` array. The length of the array should be used instead of this magic number to give more context and accuracy to the `if` statement.

Impact

Informational.

Recommendation

Use the length of the `buckets` array directly as the maximum value, because this array is what will be iterated over.

```
- if (_nBuckets > 65535) revert CommonEventsAndErrors.InvalidParam();
+ if (_nBuckets > buckets.length) revert CommonEventsAndErrors.InvalidParam();
```

Developer Response

Recommendation implemented and tests updated in PR #842

9. Informational - Complexity reduction possible

Combining multiple lines of logic can simplify `_withdrawFromBaseStrategy()`.

Technical Details

Multiple lines of code can be combined to reduce overall complexity.

Impact

Informational.

Recommendation

Consider removing unchecked and making the code more elegant by removing the intermediate `_directTransferAmount` variable:

```
-    uint256 _directTransferAmount = _balance > amount ? amount : _balance;
    unchecked {
-        _withdrawFromBaseStrategyAmount = amount - _directTransferAmount;
+        _withdrawFromBaseStrategyAmount = _balance > amount ? 0 : amount - _balance;
    }
```

Developer Response

Recommendation implemented and tests updated in PR #842.

10. Informational - Hypothetical overflow

Arithmetic in an unchecked block may overflow because no protections exist around the values summed together.

Technical Details

There is no logic that prevents this summation from overflowing in `_withdrawFromBaseStrategy()`. However, it is very unlikely for token balances to reach such a high value, so this would be safe in most normal cases.

Impact

Informational.

Recommendation

Consider removing unchecked to use SafeMath and revert in case of an overflow.

Developer Response

Recommendation implemented in PR #842.

11. Informational - `_withdrawFromBaseStrategy()` does not check if transfer may revert

The last line of code in `_withdrawFromBaseStrategy()` is a `safeTransfer()` which could revert. There are many checks throughout the TRV contract that check for incorrect cases to emit an error, but this case does not have any checks to throw an error.

Technical Details

A comment in `_withdrawFromBaseStrategy()` indicates that the amount that is intended to be withdrawn from the base strategy may not be the amount that the base strategy returns. Specifically, the DsrBaseStrategy may return fewer tokens than requested if it does not have the amount requested.

Impact

Informational.

Recommendation

Consider adding a check to revert if `amount > token.balanceOf(address(this))`. There is no need to add a check if gas savings is a priority and there is an implied assumption that the amount argument chosen by the multisig will always be chosen to avoid a revert.

Developer Response

Recommendation implemented and tests updated in PR #842.

12. Informational - `totalAvailable()` can be declared external

`totalAvailable()` is a public function but is never called internally, so it can be an external function instead of a public function.

Technical Details

`totalAvailable()` is a public function that does not need to be public.

Impact

Informational.

Recommendation

Modify `totalAvailable()` to an external function.

Developer Response

Recommendation implemented in PR #842.

13. Informational - Typos

Some variable names have typos and some comments are inaccurate.

Technical Details

1. In AuraStaking.sol: “_recipeint” -> “_recipient”
2. In TreasuryReservesVault.sol, `recoverToken()` was copied from TempleDebtToken.sol but the comment was not modified: “Recover any token from the debt token” -> “Recover any token from the TRV”
3. In TreasuryReservesVault.sol: “mroe” -> “more”
4. In TreasuryReservesVault.sol:: “dToken’s” -> “dTokens”
5. In AbstractStrategy.sol and the 5 other strategy contracts: “stratgy” -> “strategy”
6. In AbstractStrategy.sol and the 5 other strategy contracts: “onthe” -> “on the”
7. In TempleDebtToken.sol: “stoarge” -> “storage”
8. The link in this comment is broken

Impact

Informational.

Recommendation

Fix typos.

Developer Response

Recommendation implemented in PR #842

14. Informational - Usage of `onlyElevatedAccess` modifier in public functions

Access protection can be accidentally bypassed when calling public functions internally.

Technical Details

The implementation of the `onlyElevatedAccess()` modifier relies on `msg.sig` to authorize access to a particular function. This variable refers to the first 4 bytes of the calldata, which contains the intended function selector to be executed on the called contract. The TempleElevatedAccess.sol contract holds a mapping that defines which callers have access to which functions (represented by their function selectors).

It may be possible to ignore the restrictions if this modifier is used in public functions. Consider the following example contract:

```
contract SomeContract is TempleElevatedAccess {
  function foo() external onlyElevatedAccess {
    ...
    bar();
    ...
  }

  function bar() public onlyElevatedAccess {
    ...
  }
}
```

If `bar()` is called externally then `onlyElevatedAccess` will correctly check the authorization using the function selector for `bar()`. But if `bar()` is called internally, like in the case of `foo()`, then the check will be done using the function selector of the original call to the contract (`foo()`), as `msg.sig` will still refer to the first four bytes of the calldata.

Impact

Informational.

Recommendation

The codebase doesn't contain any instance of this modifier being used in public functions. It is advised to take precaution when using this authorization solution with public functions.

Developer Response

This is understood and tests already exist to verify this. An extra comment has been added to the function to alert future protocol engineers of the risk.

15. Informational - TLC design may increase liquidation risk

Some design decisions in the Temple Line of Credit (TLC) may increase the risk of user liquidations in cases where lenders have a high loan-to-value (LTV) loan.

Technical Details

Unlike protocols like Compound Finance, there is no difference between the max LTV that a user can borrow and the liquidation threshold. When combined with the interest that accumulates on the loan each block, this means that a loan with 85% LTV can be liquidated in the next block.

Another scenario that can cause user liquidation is reducing the Treasury Price Index (TPI). Because the value of TEMPLE collateral is price using the TPI, when the TPI drops, this causes an immediate drop in the value of collateral in TLC and cause positions to be eligible for liquidation. In normal market conditions, a reduction in TPI is unlikely, and a liquidation is unlikely to induce bad debt, but in the worst case scenario, governance's choice to reduce TPI can trigger liquidations that lead to bad debt for the protocol.

Impact

Informational.

Recommendation

Several choices can be made to reduce the risk of user liquidations:

1. Consider adding a buffer between the max LTV and the liquidation threshold. For example, the Compound v2 DAI market on mainnet has a max LTV of X and a liquidation threshold of X+Y. The TLC frontend should not allow loans with max LTV, and should provide clear warnings or risk indicators when taking out a loan above, say, 60% or 70% LTV.
2. Whenever the TPI is scheduled to be lowered, users should be notified that the pending change could cause liquidations for high risk positions. This is an example of Compound Finance doing something similar when they lower collateral factors.
3. TempleDAO documentation should clarify that it is possible, though unlikely, for TPI to decline. A user inferring from historic data may say that TPI is an up-only number, which would make it look very safe as collateral. But there is nothing preventing the possibility of a TPI decrease if treasury circumstances dictate that it should happen.
4. Consider adding TempleDAO to the Risk DAO bad debt dashboard and use a Telegram bot to allow key stakeholders and governance to monitor for changes in the bad debt level of the protocol.

Developer Response

The UI implementation will NOT let users borrow right up to the max LTV, there will be a buffer implemented at that layer. Eg if (hypothetically) the max LTV is 80%, the UI may only let users borrow up to say 75%. If users interact with the contracts directly (or via etherscan) then it is assumed they are power users and know what they are doing.

An extra comment/warning has been added to the `borrow()` function to advise this in PR #842

Extra documentation in gitbook will be added when TLC is released to mainnet, and adding to the bad debt dashboards/TG bots is also a good suggestion.

Final remarks

This TempleDAO protocol involves many strategies interacting with a core vault and a unique debt token accounting feature to account for opportunity cost of any strategy. The accounting throughout this system may be the most complex component of the system, but few errors were found in the accounting logic. The TLC borrowing is one of the few important features in these contracts that is not protected by a modifier, and therefore it is of special importance.

The scope of the audit was focused on the on-chain elements, but many key design elements are not found on-chain or in solidity code. For example, the TPI oracle price is set and rebalanced with an off-chain bot which we had no visibility into. This bot plays a key role in setting crucial value parameters, so the security around its implementation should also be considered. Other critical off-chain elements in the TempleDAO protocol include governance actions and governance assumptions. The governance of TempleDAO is assumed to be trusted, otherwise protocol funds are at risk of getting rugged. The existing governance of TempleDAO v1, what was used at the time of this review, will be changed to introduce two brand new multisigs to replace the existing multisig. These multisigs were not created yet so there was no visibility into the choices made around this multisig. Additionally, the TEMPLE token price target, TPI, is set by governance in a very specific way with certain assumptions. For example, the price of DAI in the TPI calculations is assumed to be equal to $1. The actual TPI set in the protocol is slightly lower than the actual TPI calculated using the value of funds held by treasury, which gives the treasury a small buffer of funds which could cover bad debt or other unexpected edge cases. This knowledge and other the assumptions related to the protocol design were not clearly documented in the code or comments, but were learned through discussions with the devs.

Consider testing for undocumented invariants in the system. Some invariant examples include:

1. The interest accumulated by the DSR base strategy should equal the interest accumulated by the Maker DSR pot.
2. The interest accumulated by the stablecoin Temple Debt Token should equal the interest accumulated by the Maker DSR pot.
3. A strategy should never have credit and debt at the same time, only one of the two, otherwise the assumption in TempleLineOfCredit.sol that `_cache.trvDebtCeiling` is the maximum debt that a strategy can have is incorrect because it does not account for credits.