yAudit TempleDAO Lending Recheck 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 2 days. The code review was performed by 2 auditors between August 24 and August 25, 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 7ed36cb8a1f42145ba66f70e31c51a3f774d6bed for the TempleDAO repo. Because this was the second time that yAudit was reviewing this code, the primary focus of this audit was a review of the mitigation changes addressing issues from the original full audit by yAudit. These changes are found in PR #842.

TempleDAO fixed the issues from the first report in pull request #842 and yAudit did a review of these mitigations by comparing changes made from commit b015cd5d1df122ad5fbe0f94fb5bd070db27e335, which was the commit of the first review, to commit 7ed36cb8a1f42145ba66f70e31c51a3f774d6bed, which was the commit that included the mitigations.

Scope

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

- https://github.com/TempleDAO/temple/tree/7ed36cb8a1f42145ba66f70e31c51a3f774d6bed/protocol/contracts/v2/*
- https://github.com/TempleDAO/temple/tree/7ed36cb8a1f42145ba66f70e31c51a3f774d6bed/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 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

Category	Mark	Description
		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

None.

Medium Findings

None.

Low Findings

1. Low - Rounded down principal could be carried over in debtor cache struct

Under specific circumstances, a rounded down value for the debtor principal present in the cache structure could be carried into further calculations.

Technical Details

The `_initDebtorCache()` function is in charge of building the cache structure associated with a debtor. While calculating the `baseInterest` value, the function may eventually overwrite the principal:

```
// Because the shares => debt conversion is rounded down in some scenarios,
// there is a chance the `_debtor.principal` outstanding may be
// less than this rounded down number. So use the minimum of the two.
_debtorCache.principal =  _debtorPrincipalAndBaseInterest < _debtorCache.principal
    ? _debtorPrincipalAndBaseInterest
    : _debtorCache.principal;

// The baseInterest outstanding for this debtor is now the difference.
unchecked {
    _debtorCache.baseInterest = _debtorPrincipalAndBaseInterest - _debtorCache.principal;
}
```

As hinted in the comment from the snippet, `_debtorPrincipalAndBaseInterest` may be lower than the actual principal due to rounding issues. The solution present in the implementation is to take the minimum of these values so that the subtraction to calculate `_debtorCache.baseInterest` doesn't overflow.

While this sounds reasonable (the base interest value would end up being zero in such case), it is important to note that the principal value present in the cache struct is overwritten with a value that is actually lower than the original principal, which may impact other calculations that use the value from the struct. For example, `mint()` initializes the debtor cache and then uses the principal stored in the cache struct to update the principal value in storage.

Impact

Low. This represents an edge case and the impact is minimal as the difference would just be a rounding error.

Recommendation

Instead of overwriting the principal in the debtor cache, calculate the subtraction using `_subFloorZero()` which will handle the case when `_debtorPrincipalAndBaseInterest` is lower than `_debtorCache.principal`.

```
_debtorCache.baseInterest = _subFloorZero(_debtorPrincipalAndBaseInterest,
_debtorCache.principal);
```

Developer Response

Suggestion implemented in e01db3d

2. Low - Enabled borrow tokens configuration is not cleared when shutting down a strategy

The recently introduced set of enabled borrow tokens associated with a strategy is not cleared when the strategy is shut down and removed from the TreasuryReservesVault.sol contract.

Technical Details

The implementation of the `shutdown()` function, which is in charge of removing a strategy from the vault, fails to clear the configuration associated with the enabled borrow tokens of a strategy.

Impact

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

Recommendation

Clear the `enabledBorrowTokens` mapping in the implementation of `shutdown()`.

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

```
+   delete _strategyConfig.enabledBorrowTokens[_token];
    delete credits[_token];
```

Developer Response

Suggestion implemented in fbab4b2c

Gas Saving Findings

1. Gas - G7 `treasuryReservesVault` optimization not implemented in DsrBaseStrategy

Finding G7 in the original report had many gas optimization suggestions. One of the suggestions was not implemented but can be.

Technical Details

The suggestion to cache `treasuryReservesVault` in DsrBaseStrategy::trvWithdraw() was not applied but is still applicable.

Impact

Gas savings.

Recommendation

Implement suggested mitigation.

Developer Response

Suggestion implemented in 08a65e0b

2. Gas - Change visibility of public constants

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

Technical Details

This was originally reported as G-06 in the first audit report.

Most of the occurrences have been resolved, except the one in AbstractStrategy.sol.

The `VERSION` constant in ThresholdSafeGuard.sol is also marked as `public`, although this contract doesn't have a separate getter that publishes the constant, as the others do.

Impact

Gas savings.

Recommendation

Change the visibility of the constants to `private`.

Developer Response

Suggestion implemented in 94fe2844

3. Gas - Avoid copying calldata to memory in `addStrategy()`

In the `addStrategy()` function, the elements of the `debtCeiling` array are copied from calldata to memory in each iteration of the loop that traverses the array.

Technical Details

Line 184 defines a variable in memory that is used to assign each element of the `debtCeiling` array, which will end up copying the `AssetBalance` structure from calldata to memory.

Impact

Gas savings.

Recommendation

Alias the variable using calldata instead of copying it to memory:

```
-  ITempleStrategy.AssetBalance memory _assetBalance;
+  ITempleStrategy.AssetBalance calldata _assetBalance;
```

Developer Response

Suggestion implemented in 36f0347d

Informational Findings

1. Informational - Imprecise comment

Comments for similar caching function are different, and one of the comment may be slightly misleading.

Technical Details

`_debtTokenCache()`, `_getDebtorCache()`, and `_getBaseCache()` are similar functions in different contracts. The NatSpec describing `_debtTokenCache()` is `Update storage if and only if the state has changed.` while the NatSpec for `_getBaseCache()` and `_getDebtorCache()` is `Update`

`storage if and only if the timestamp has changed since last time.` In reality, the functions rely on a timestamp if statement check, not specifically a state change check. The timestamp comment is more accurate and can be applied to both functions.

Additional alignment between related caching functions is possible. `_initDebtTokenCache()` checks if `blockTs != interestAccumulatorUpdatedAt` while `_initBaseCache()` and `_initDebtorCache()` checks if `_timeElapsed > 0`. Aligning the logic in related functions can reduce overall protocol complexity.

Impact

Informational.

Recommendation

Improve alignment between the comments and design of caching functions in different contracts.

Developer Response

Suggestions implemented in 3ef74793

2. Informational - Potential overflow in `_initDebtorCache()`

The principal and risk premium are added together using `unchecked` math, which may result in an overflow for large values that exceed the 128 bit range when combined.

Technical Details

In `_initDebtorCache()` the debtor's principal and current risk premium are added together to calculate the updated risk premium interest.

Since the data type of the resulting expression (`_debtorTotalDue`) is of type `uint128` (same as the operands), the calculation may silently overflow due to the usage of `unchecked` math.

Impact

Informational.

Recommendation

Change `_debtorTotalDue` to `uint256` to allow a temporary expansion of the data type precision to calculate the updated compounded interest. The calculation can be then casted back to `uint128` after the principal is subtracted.

```
// Calculate the new amount of risk premium interest by compounding the total debt
// and then subtracting just the principal.
uint256 _debtorTotalDue;
unchecked {
    _debtorTotalDue = _debtorCache.principal + _debtorCache.riskPremiumInterest;
}
_debtorTotalDue = _debtorTotalDue.continuouslyCompounded(
    _timeElapsed,
    _debtorCache.riskPremiumRate
);

unchecked {
    uint128 _newRiskPremiumInterest = (_debtorTotalDue -
_debtorCache.principal).encodeUInt128();
    _debtorCache.riskPremiumInterestDelta = _newRiskPremiumInterest -
_debtorCache.riskPremiumInterest;
    _debtorCache.riskPremiumInterest = _newRiskPremiumInterest;
}
```

Developer Response

Suggestions implemented in 27d521cf

Note the above suggestion should cast `_debtorCache.principal` as a uint256 within the unchecked block. Without it, the upcast would only happen on the result - so it could still overflow.

```
unchecked {
    _debtorTotalDue = uint256(_debtorCache.principal) + _debtorCache.riskPremiumInterest;
}
```

3. Informational - Unchecked statement could overflow

Finding G10 in the original report pointed out specific lines of code that could be made unchecked due to a lack of overflow or underflow risk. One additional line of code not mention in G10 is in an unchecked statement that could hypothetically overflow. The value is only an

temporary variable, not a state variable, so the only risk is misuse of the event that emits this value.

Technical Details

A summation in `_mintDToken()` newly moved to an unchecked block could hypothetically overflow. The comment before this line hints that the overflow is `checked within the dToken.mint()`, but this is incorrect. `dToken.mint()` does not check for an overflow of `_newDebt + dTokenBalance`, where `dTokenBalance` is the sum of principal, baseInterest, and riskPremiumInterest. The closest that `dToken.mint()` achieves is confirming no overflow in the summation `_debtorCache.principal + _mintAmountUInt128`, but this summation omits the baseInterest and riskPremiumInterest values that are included in `dTokenBalance`.

Impact

Informational.

Recommendation

Remove this line from the unchecked block to prevent an overflow.

Developer Response

Rather than only checking in TRV that the dToken balance doesn't overflow, in my opinion it would be best to add the overflow check into the `dToken.mint()` itself, to match the comment. This means if we add mint/burn capabilities elsewhere it will also be checked rather than only in the TRV.

This has been implemented by tracking the totalBalance in the dToken's `DebtorCache`

Implemented in 612ead34

4. Informational - Code without math operations doesn't need unchecked

G10 pointed out gas optimization by moving some lines of code with math operations into unchecked blocks. If there is no math operation, it makes no difference whether the code is in an unchecked block or not, but generally only unchecked math operations are placed inside unchecked blocks.

Technical Details

Consider moving the line `if (_delta > maxTreasuryPriceIndexDelta) revert BreachedMaxTpiDelta(_oldTpi, _newTpi, maxTreasuryPriceIndexDelta)` outside of this unchecked block.

The same suggestion applies to `debtTokenData.totalDebt = _cache.totalDebt = _newDebt;` in `_repayTotalDebt()`.

Impact

Informational.

Recommendation

Generally unchecked blocks only include lines of code involving math operations that may overflow, so move other lines of code outside the unchecked blocks.

Developer Response

Suggestion won't be implemented. In my opinion, this recommendation would end up using more gas, since the variable (eg `uint256 _delta;`) would have to be declared (and set to 0) outside of the unchecked block first, meaning another EVM instruction.

Given there's no functional difference with this approach, there's not a compelling reason to update.

5. Informational - `currentTotalDebt()` can be declared external

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

Technical Details

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

Impact

Informational.

Recommendation

Modify `currentTotalDebt()` to an external function.

Developer Response

Suggestion implemented in 1bed04a2

6. Informational - `_withdrawFromBaseStrategy()` should also revert explicitly if withdrawn amount from base strategy is not enough

In order to mitigate INFO-11 of the original audit report, an explicit revert was added to `_withdrawFromBaseStrategy()` to check if the available amount is enough to cover the requested withdrawal amount. However, this check fails to cover all potential scenarios.

Technical Details

The check and the revert was added in line 624 which only covers the `else` branch of the conditional present in line 591.

If there is a base strategy associated with the debt token, it is possible that this strategy may not have the required funds and could potentially return an amount less than what was requested. In such a scenario, the condition may fail to be correctly checked.

Impact

Informational.

Recommendation

Move the check out of the `else` branch so that it also applies to the case when there is a base strategy associated with the borrow token.

Developer Response

Suggestion implemented in d952e91f

7. Informational - Emit `RewardTokensSet` event in AuraStaking.sol constructor

The constructor of the AuraStaking.sol contract should also emit the `RewardTokensSet` event to signal the assignment of the array of reward tokens.

Technical Details

Given the inclusion of the `setRewardTokens()` function, which now emits the `RewardTokensSet` event when the reward tokens array is updated, the constructor of the contract should also emit the same event since it performs an assignment of the same array.

Impact

Informational.

Recommendation

Emit the `RewardTokensSet` event in the constructor of AuraStaking.sol.

```
constructor(
    address _initialRescuer,
    address _initialExecutor,
    IERC20 _bptToken,
    IAuraBooster _booster,
    address[] memory _rewardTokens
```

```
  ) TempleElevatedAccess(_initialRescuer, _initialExecutor)
  {
      bptToken = _bptToken;
      booster = _booster;
      rewardTokens = _rewardTokens;
+     emit RewardTokensSet(_rewardTokens);
  }
```

Developer Response

Suggestion won't be implemented. We don't feel the need to emit events within the constructor (generally speaking) given it is a one-time setup. Events are primiarily reserved for when values are modified after construction. Doing so here would be inconsistent with most of the other contracts.

8. Informational - Strategy details could output debt ceiling only for enabled tokens

View functions present in the TreasuryReservesVault.sol contract output strategy information for all tokens in the set of borrow tokens, which may not necessarily be the same set of enabled tokens for the strategy.

Technical Details

Both `strategyDetails()` and `strategyBalanceSheet()` return information based on all tokens present in the configured set of borrow tokens (i.e. `_borrowTokenSet`).

With the introduction of the enabled borrow tokens feature for each particular strategy, these results could be bound to only the enabled tokens instead of the whole set.

Impact

Informational.

Recommendation

Restrict the results to the set of enabled borrow tokens for the given strategy.

Developer Response

Suggestion won't be implemented. There won't be a very large number of enabled borrow tokens (< 5), and these functions should also handle the case where a borrow token was enabled for a strategy, which was later disabled.

Further, the logic to implement this would become more complicated, as the number of `enabledBorrowTokens` for this strategy would have to be calculated in a separate iteration first.

Given the default balances will be 0 and the low number of borrow tokens, the implementation is ok as is.

9. Informational - Debt ceiling hook is not triggered when a strategy is added to the vault

The mitigation for issue M-02 introduced the concept of a hook that gets triggered when the debt ceiling for a strategy is changed in the TreasuryReservesVault.sol contract. While this callback is correctly fired when the debt ceiling is updated, it is not triggered when the strategy is first added and its debt ceiling is initialized.

Technical Details

The `setStrategyDebtCeiling()` function fires the `debtCeilingUpdated()` callback to signal the strategy that its associated debt ceiling has been updated.

The same hook is not triggered in the `addStrategy()` function, which initializes the debt ceiling value for each of the configured tokens for the strategy. Considering the strategy contract is created before being added to the vault, the strategy might not be aware about the eventual configuration and could potentially require this value during its setup.

Impact

Informational.

Recommendation

Although there is no impact in any existing strategy, it could be useful to add the trigger to `addStrategy()` to make the codebase future proof.

Developer Response

Suggestion won't be implemented. This idea was originally considered while fixing M-02, however it doesn't much provide value as there's frequently going to be a 'chicken and egg' problem when setting up a new strategy. For example with TLC, the hook would be to update the interest rate. However this cannot be updated until the TRV is set on TLC (since it needs the debt ceiling for the utilisation rate).

Instead, this is intentionally left to the strategy to decide if it needs to do any post-create steps. For TLC, `setTlcStrategy()` explicitly calls `_updateInterestRates()`

Final remarks

The issues raised in the first audit report were successfully mitigated, as evidence by the lack of any findings above the low risk categorization. There were some very minor improvements that could be made, but no substantial risks were observed during the audit.

The changeset also included an important refactor to the TempleDebtToken.sol contract to introduce the “cache struct” pattern, already employed in the TempleLineOfCredit.sol contract. This set of modifications were not a suggestion from the previous yAudit report, but were grouped in with the other changes during the audit. With the intention of lowering gas costs, this pattern initializes a shared structure that pre-fetches storage variables that are frequently accessed in the course of different functions to have them cached in a structure stored in memory. While this approach effectively reduces gas consumption, it is important to note that it adds complexity while reasoning about the contract’s functionality, and could be more error prone compared to direct storage access. It is also fair to mention that, in some cases, code complexity was reduced due to having certain calculations already available in the cache structure. Such an example is the `_burn()` function, which is now simpler and easier to read.

One suggestion to streamline a similar future audit that is focused on mitigations is to better correlate specific commit hashes or smaller PRs to every issue in the original audit report. The approach used prior to this audit was to group all changes into a single PR, which is also useful, but improving the developer response comments under each issue in the original report to point directly at the changes made only to address that specific issue could have reduced the effort needed to validate the mitigations.