// Security Assessment

06.12.2025 - 06.13.2025

AppTreasury Contract

Rezerve Money

HALBORN

AppTreasury Contract - Rezerve Money

Prepared by: HALBORN

Last Updated 06/19/2025

Date of Engagement: June 12th, 2025 - June 13th, 2025

Summary

100% ⓘ OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
14	0	0	0	6	8

TABLE OF CONTENTS

8. Automated Testing

1. Summary

`Rezerve Money` engaged `Halborn` to conduct a security assessment of their `AppTreasury contract` beginning on June 11th, 2025 and ending on June 12th, 2025. The security assessment was scoped to the smart contract provided in the GitHub repository. Commit hash and further details can be found in the Scope section of this report.

AppTreasury is a fork of OlympusDAO's treasury with improved logic and monetary policies. The contact was upgraded to a recent solidity version and modified to be used using proxies. Credit and debit functionalities were added to allow for features such as PSM and staking rewards to later on come into the picture.

2. Assessment Summary

`Halborn` was provided 2 days for the engagement and assigned one full-time security engineer to review the security of the smart contract in scope. The engineer is blockchain and smart contract security expert with advanced penetration testing and smart contract hacking skills, and deep knowledge of multiple blockchain protocols.

The purpose of the assessment is to:

- Identify potential security issues within the smart contract.
- Ensure that smart contract functionality operates as intended.

In summary, Halborn identified some improvements to reduce the likelihood and impact of risks, which were partially addressed by the `Rezerve Money team`. The main ones were the following:

- `In the disable() function, consider removing the _toDisable address from the tokens array to be consistent with the enabledTokens mapping.`
- `Make sure all inherited upgradable contracts are initialized.`
- `Either add support to fee-on-transfer tokens or document that fee-on-transfer tokens are not supported.`
- `Implement proper input validation in all functions.`
- `Consider adding a constructor and calling the _disableinitializers() method inside.`
- `Use the initializer modifier for the initial setup instead of reinitializer.`

3. Test Approach And Methodology

`Halborn` performed a combination of manual and automated security testing to balance efficiency, timeliness, practicality, and accuracy in regard to the scope of this assessment. While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance coverage of the code and can quickly identify items that do not follow the security best practices. The following phases and associated tools were used during the assessment:

- Research into architecture and purpose.
- Smart contract manual code review and walkthrough.
- Graphing out functionality and contract logic/connectivity/functions (`solgraph`).
- Manual assessment of use and safety for the critical Solidity variables and functions in scope to identify any arithmetic related vulnerability classes.
- Manual testing by custom scripts.
- Static Analysis of security for scoped contract, and imported functions (`slither`).

4. RISK METHODOLOGY

Every vulnerability and issue observed by Halborn is ranked based on **two sets** of **Metrics** and a **Severity Coefficient**. This system is inspired by the industry standard Common Vulnerability Scoring System.

The two **Metric sets** are: **Exploitability** and **Impact**. **Exploitability** captures the ease and technical means by which vulnerabilities can be exploited and **Impact** describes the consequences of a successful exploit.

The **Severity Coefficients** is designed to further refine the accuracy of the ranking with two factors: **Reversibility** and **Scope**. These capture the impact of the vulnerability on the environment as well as the number of users and smart contracts affected.

The final score is a value between 0-10 rounded up to 1 decimal place and 10 corresponding to the highest security risk. This provides an objective and accurate rating of the severity of security vulnerabilities in smart contracts.

The system is designed to assist in identifying and prioritizing vulnerabilities based on their level of risk to address the most critical issues in a timely manner.

4.1 EXPLOITABILITY

ATTACK ORIGIN (AO):

Captures whether the attack requires compromising a specific account.

ATTACK COST (AC):

Captures the cost of exploiting the vulnerability incurred by the attacker relative to sending a single transaction on the relevant blockchain. Includes but is not limited to financial and computational cost.

ATTACK COMPLEXITY (AX):

Describes the conditions beyond the attacker's control that must exist in order to exploit the vulnerability. Includes but is not limited to macro situation, available third-party liquidity and regulatory challenges.

METRICS:

EXPLOITABILITY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Origin (AO)	Arbitrary (AO:A) Specific (AO:S)	1 0.2
Attack Cost (AC)	Low (AC:L) Medium (AC:M) High (AC:H)	1 0.67 0.33

EXPLOITABILITY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Complexity (AX)	Low (AX:L) Medium (AX:M) High (AX:H)	1 0.67 0.33

Exploitability E is calculated using the following formula:

$$E = \prod m_e$$

4.2 IMPACT

CONFIDENTIALITY (C):

Measures the impact to the confidentiality of the information resources managed by the contract due to a successfully exploited vulnerability. Confidentiality refers to limiting access to authorized users only.

INTEGRITY (I):

Measures the impact to integrity of a successfully exploited vulnerability. Integrity refers to the trustworthiness and veracity of data stored and/or processed on-chain. Integrity impact directly affecting Deposit or Yield records is excluded.

AVAILABILITY (A):

Measures the impact to the availability of the impacted component resulting from a successfully exploited vulnerability. This metric refers to smart contract features and functionality, not state. Availability impact directly affecting Deposit or Yield is excluded.

DEPOSIT (D):

Measures the impact to the deposits made to the contract by either users or owners.

YIELD (Y):

Measures the impact to the yield generated by the contract for either users or owners.

METRICS:

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
Confidentiality (C)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
Integrity (I)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1
Availability (A)	None (A:N) Low (A:L) Medium (A:M) High (A:H) Critical (A:C)	0 0.25 0.5 0.75 1
Deposit (D)	None (D:N) Low (D:L) Medium (D:M) High (D:H) Critical (D:C)	0 0.25 0.5 0.75 1
Yield (Y)	None (Y:N) Low (Y:L) Medium (Y:M) High (Y:H) Critical (Y:C)	0 0.25 0.5 0.75 1

Impact I is calculated using the following formula:

$$I = max(m_I) + \frac{\sum m_I - max(m_I)}{4}$$

4.3 SEVERITY COEFFICIENT

REVERSIBILITY (R):

Describes the share of the exploited vulnerability effects that can be reversed. For upgradeable contracts, assume the contract private key is available.

SCOPE (S):

Captures whether a vulnerability in one vulnerable contract impacts resources in other contracts.

METRICS:

SEVERITY COEFFICIENT (C)	COEFFICIENT VALUE	NUMERICAL VALUE
Reversibility (r)	None (R:N) Partial (R:P) Full (R:F)	1 0.5 0.25
Scope (s)	Changed (S:C) Unchanged (S:U)	1.25 1

Severity Coefficient C is obtained by the following product:

$$C = rs$$

The Vulnerability Severity Score S is obtained by:

$$S = min(10, EIC * 10)$$

The score is rounded up to 1 decimal places.

SEVERITY	SCORE VALUE RANGE
Critical	9 - 10
High	7 - 8.9
Medium	4.5 - 6.9
Low	2 - 4.4
Informational	0 - 1.9

5. SCOPE

FILES AND REPOSITORY

(a) Repository: code

(b) Assessed Commit ID: dfc77dd

(c) Items in scope:

- contracts/AppTreasury.sol

Out-of-Scope: Third party dependencies and economic attacks.

REMEDIATION COMMIT ID:

- 4c4d2c9
- cb40ced
- cd73e8e

Out-of-Scope: New features/implementations after the remediation commit IDs.

6. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW
0	0	0	6

INFORMATIONAL
8

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
INCOMPLETE DISABLE() FUNCTION	LOW	SOLVED - 06/12/2025

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
MISSING INITIALIZATION OF INHERITED UPGRADABLE CONTRACTS	LOW	SOLVED - 06/15/2025
POTENTIAL INCOMPATIBILITIES WITH FEE-ON-TRANSFER TOKENS	LOW	RISK ACCEPTED - 06/18/2025
MISSING INPUT VALIDATION	LOW	RISK ACCEPTED - 06/18/2025
MISSING _DISABLEINITIALIZERS() CALL IN THE CONSTRUCTOR	LOW	RISK ACCEPTED - 06/18/2025
MISUSE OF REINITIALIZER AND MISSING ONLYINITIALIZING ON SUBINITIALIZER	LOW	RISK ACCEPTED - 06/18/2025
INEFFICIENT ENABLED TOKENS LOGIC	INFORMATIONAL	ACKNOWLEDGED - 06/18/2025
UNLOCKED PRAGMA COMPILER	INFORMATIONAL	SOLVED - 06/14/2025
USE OF REVERT STRINGS INSTEAD OF CUSTOM ERRORS	INFORMATIONAL	ACKNOWLEDGED - 06/18/2025
STYLE GUIDE OPTIMIZATIONS	INFORMATIONAL	ACKNOWLEDGED - 06/18/2025
CONSIDER USING NAMED MAPPINGS	INFORMATIONAL	SOLVED - 06/12/2025

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
UNSAFE ERC20 OPERATION IN USE	INFORMATIONAL	SOLVED - 06/15/2025
CACHE ARRAY LENGTH OUTSIDE OF LOOP	INFORMATIONAL	ACKNOWLEDGED - 06/18/2025
INCONSISTENT ERROR MESSAGES	INFORMATIONAL	ACKNOWLEDGED - 06/18/2025

7. FINDINGS & TECH DETAILS

7.1 INCOMPLETE DISABLE() FUNCTION

// LOW

Description

The `disable()` function only sets `enabledTokens[_toDisable]` back to `false`, and does not remove the `_toDisable` address from the `tokens` array. As a result, the `tokens` array retains addresses of disabled reserve assets, creating a mismatch between the `tokens` array and the `enabledTokens` mapping. Notice every invocation of `calculateActualReserves()` or `_updateReserves()` would iterate over the (ever-growing) `tokens` array, but skips disabled entries via the mapping, nonetheless still paying the full gas cost of the loop. Over time, each disable operation thus "bloats" the array with stale entries, risking a gas-denial-of-service if the list grows large enough to exceed block gas limits.

Moreover, front-end dashboards and off-chain analytics could call `tokens()` to enumerate treasury assets; disabled tokens will still appear in such lists, leading to misleading UI data and potential misinformed governance or integration decisions. This pattern of stale state mirrors "state inconsistency" or "state derailment" defects seen in DEX contracts, where outdated or unused entries persist in critical data structures, causing both confusion and exploitable logic gaps.

Finally, notice it is possible to disable a token even when it is used as reserve.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:L/I:L/D:N/Y:N (3.1)

Recommendation

In the `disable()` function, consider removing the `_toDisable` address from the `tokens` array to be consistent with the `enabledTokens` mapping.
Finally, decide and document the situation where a token used as reserve is disabled.

Remediation Comment

SOLVED: The **Reserve Money team** fixed this finding in commit `4c4d2c9` by removing the `enabledTokens` mapping and `tokens` array of the contract and started using `EnumerableSet`.

Remediation Hash

https://github.com/rezervemoney/code/commit/4c4d2c9d57875b28a7e9abe545bde88497216fe3

7.2 MISSING INITIALIZATION OF INHERITED UPGRADABLE CONTRACTS

// LOW

Description

The `AppTreasury` contract inherits from multiple modules like `AppAccessControlled`, `IAppTreasury`, `PausableUpgradeable` and `ReentrancyGuardUpgradeable` but only calls the initializers for `AppAccessControlled` (`__AppAccessControlled_init()`) and `PausableUpgradeable` (`__Pausable_init()`) within the `initialize()` function. This leaves `ReentrancyGuardUpgradeable` uninitialized.

In the OpenZeppelin upgradeable contract pattern, initializers are used to set up state variables and other configurations that would typically be done in a constructor for non-upgradeable contracts. Failing to explicitly call the `__ReentrancyGuard_init()` initializer means that any future changes to the `ReentrancyGuardUpgradeable` contract's initialization logic will not be applied to this contract.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:L/D:N/Y:N (2.5)

Recommendation

It is recommended to initialize all inherited upgradable contracts. More specifically, explicitly call the `__ReentrancyGuard_init()` function in the `initialize()` function to ensure all inherited modules are properly initialized. This approach aligns with standard practices for using the OpenZeppelin upgradeable pattern and helps future-proof the contract against potential changes in the underlying library.

Remediation Comment

SOLVED: The **Reserve Money team** fixed this finding in commit `cb40ced` by explicitly calling the `__ReentrancyGuard_init()` function in the `initialize()` function as recommended.

Remediation Hash

https://github.com/rezervemoney/code/commit/cb40ced1158c1c410582e446244ea64200fd25a2

7.3 POTENTIAL INCOMPATIBILITIES WITH FEE-ON-TRANSFER TOKENS

// LOW

Description

The `AppTreasury` contract assumes that calling `safeTransferFrom()` always results in exactly `_amount` tokens arriving, but **fee-on-transfer** tokens deduct a fee on each transfer, so the actual amount received can be less than `_amoun` . Consequently, `tokenValueE18(_token, _amount)` could overestimate the treasury's reserves and causes **over-minting** of the protocol's token, leading to inaccurate accounting and potential inflation.

The contract assumes that the amount specified in transfer operations equals the amount actually received. This assumption is visible in the `deposit()` function:

```
54  function deposit(uint256 _amount, address _token, uint256 _profit)
55    external
56    override
57    nonReentrant
58    whenNotPaused
59    onlyReserveDepositor
60    returns (uint256 send_)
61  {
62    require(enabledTokens[_token], "Treasury: invalid token");
63
64    IERC20(_token).safeTransferFrom(msg.sender, address(this), _amount);
65
66    uint256 value = tokenValueE18(_token, _amount);
67
68    // mint app needed and store amount of rewards for distribution
69    send_ = value - _profit;
70    app.mint(msg.sender, send_);
71
72    _totalReserves += value;
73    ...
```

- **Assumed vs. Actual Received:** The code uses the **requested** `_amount` to compute `value` , not the **actual** balance change, so fees charged by the token contract are ignored.
- **Over-Minting Risk:** Because the treasury mints based on the higher, assumed `value` , depositors of fee-on-transfer tokens receive more protocol tokens than warranted, inflating supply.
- **Reserve Miscalculation:** The `_totalReserves` counter increments by the assumed `value` , causing reserve tallies to exceed on-chain balances and obscuring true collateral levels.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:L/Y:N (2.5)

Recommendation

It is recommended to either add support to fee-on-transfer tokens by verifying the actual transferred amount or explicitly document in the contract that fee-on-transfer tokens are not supported.

Remediation Comment

RISK ACCEPTED: The **Reserve Money team** accepted the risk of this finding indicating that they won't use fee on transfer tokens.

7.4 MISSING INPUT VALIDATION

// LOW

Description

During the security assessment, it was identified that some functions in the smart contract lack proper input validation, allowing critical parameters to be set to undesired or unrealistic values. This can lead to potential vulnerabilities, unexpected behavior, or erroneous states within the contract. Examples include:

- The functions `deposit()`, `withdraw()` and `manage()` are not checking if the `_amount` argument is 0.
- The `_profit` argument in `deposit()` is not validated against the calculated `value`. Therefore, it could cause an immediate, low-level panic revert without a descriptive error message.
- The `initialize()` function is not verifying if the `_authority` argument is `address(0)`.

This list is not exhaustive. It is recommended to conduct a comprehensive review of the codebase to identify and assess other functions that may require additional input validation. Ensuring appropriate checks are in place for critical parameters will enhance the overall reliability, security, and predictability of the contracts.

BVSS

AO:A/AC:L/AX:M/R:N/S:U/C:N/A:N/I:L/D:L/Y:L (2.5)

Recommendation

To mitigate these issues, implement input validation in all constructor functions and other critical functions to ensure that inputs meet expected criteria. This can prevent unexpected behaviors and potential vulnerabilities.

Remediation Comment

RISK ACCEPTED: The **Reserve Money team** accepted the risk of this finding indicating that they won't use fee on transfer tokens.

7.5 MISSING _DISABLEINITIALIZERS() CALL IN THE CONSTRUCTOR

// LOW

Description

The `AppTreasury` contract follows an upgradeable pattern, indirectly inheriting from the `Initializable` module from OpenZeppelin. In order to prevent leaving the contracts uninitialized, OpenZeppelin's documentation recommends adding the `_disableInitializers()` function in the `constructor` to automatically lock the contracts when they are deployed. However, the upgradeable contract in scope is missing this function call.

This omission can lead to potential security vulnerabilities, as an uninitialized implementation contract can be taken over by an attacker, which may impact the proxy.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:L/D:N/Y:N (2.5)

Recommendation

Consider adding a constructor and calling the `_disableinitializers()` method within the contracts to prevent the implementation from being initialized.

```
constructor() {
  _disableInitializers();
}
```

Remediation Comment

RISK ACCEPTED: The **Reserve Money team** accepted the risk of this finding indicating that they won't use fee on transfer tokens.

7.6 MISUSE OF REINITIALIZER AND MISSING ONLYINITIALIZING ON SUBINITIALIZER

// LOW

Description

The `initialize()` function in `AppTreasury` is using the modifier `reinitializer(5)`, implying this is the fifth upgrade. However, this contract has no prior deployments. In OpenZeppelin's model, `initializer` corresponds to version 1, and `reinitializer(N)` is intended only for genuine Nth-version upgrades. Using `reinitializer(5)` on a fresh deployment has no effect beyond misleading readers and risking incorrect version assumptions

Furthermore, as per the `Initializable` convention, parent initialization routines called by child initializers must use the `onlyInitializing` modifier, ensuring they run only during an initialization context and cannot be invoked arbitrarily. Currently, `__AppAccessControlled_init(address)` lacks this modifier, allowing it to execute outside the intended initialization phase and potentially misconfigure access control if ever called improperly. For more information, see: https://docs.openzeppelin.com/contracts/5.x/api/proxy#Initializable-onlyInitializing-- and https://www.rareskills.io/post/initializable-solidity

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:L/D:N/Y:N (2.5)

Recommendation

- **Use the `initializer` modifier for the initial setup instead of `reinitializer`:**

```
function initialize(address _dre, address _dreOracle, address _authority) external initializer {
```

Reserve `reinitializer(N)` for each subsequent true upgrade step (e.g., `reinitializer(2)`, `reinitializer(3)`, etc.)

- **Guard Sub-Initializers with onlyInitializing:**

Update the `__AppAccessControlled_init()` function to:

```
function __AppAccessControlled_init(address _authority) internal onlyInitializing {
```

ensuring it can only be invoked within an active initializer or reinitializer context.

- **Document Initialization Versioning:**

Clearly outline in code comments and project documentation the expected initialization/versioning sequence, mapping each function to its intended version, to prevent confusion and enforce correct upgrade practices.

Remediation Comment

RISK ACCEPTED: The **Reserve Money team** accepted the risk of this finding.

7.7 INEFFICIENT ENABLED TOKENS LOGIC

// INFORMATIONAL

Description

The `enable()` function iterates through the entire `tokens` array to detect duplicates but, upon finding one, still updates the `enabledTokens` mapping and emits `TokenEnabled`, resulting in misleading events and redundant on-chain writes.

```
33  function enable(address _address) external onlyGovernor {
34    require(_address != address(0), "Zero address");
35
36    // RZR should not be enabled as a reserve; as this creates a circular dependency
37    require(_address != address(app), "RZR address");
38
39    // add token into tokens array if not already added
40    bool isAdded = false;
41    for (uint256 i = 0; i < tokens.length; i++) {
42      if (tokens[i] == _address) {
43        isAdded = true;
44        break; //@audit it could just revert here indicating the token is already enabled
45      }
46    }
47    if (!isAdded) tokens.push(_address);
48
49    enabledTokens[_address] = true;
50    ...
```

Furthermore, `calculateActualReserves()` loops over every entry in `tokens`, checking `enabledTokens` each time, even though all tokens in the `tokens` list should be enabled.

```
207  function calculateActualReserves() public view override returns (uint256 reserves) {
208    for (uint256 i = 0; i < tokens.length; i++) {
209      if (enabledTokens[tokens[i]]) {
210        ...
```

BVSS

AO:A/AC:M/AX:M/R:N/S:U/C:N/A:N/I:L/D:N/Y:N (1.1)

Recommendation

- In `enable()`, immediately revert on duplicates (e.g. `revert TokenAlreadyEnabled()`) before setting the mapping or emitting an event:

```
function enable(address _address) external onlyGovernor {
  require(_address != address(0), "Zero address");
  // RZR should not be enabled as a reserve; as this creates a circular dependency
  require(_address != address(app), "RZR address");

  // add token into tokens array if not already added
  uint256 tokensLength = tokens.length;
  for (uint256 i = 0; i < tokensLength; ++i) {
    if (tokens[i] == _address) revert TokenAlreadyEnabled();
  }
  tokens.push(_address);
  enabledTokens[_address] = true;
```

- In `calculateActualReserves()`, assume the `tokens` array contains only active entries (once duplicates are barred) or filter out disabled tokens via swap-and-pop in `disable()`, so the loop need not re-check every address.

```
function calculateActualReserves() public view override returns (uint256 reserves) {
  uint256 tokensLength = tokens.length;
  for (uint256 i = 0; i < tokensLength; ++i) {
      ...
```

Remediation Comment

ACKNOWLEDGED: The **Reserve Money team** accepted the risk of this finding indicating that they won't use fee on transfer tokens.

7.8 UNLOCKED PRAGMA COMPILER

// INFORMATIONAL

Description

The contract in scope currently uses a pragma version `^0.8.15`, which means that the code can be compiled by any compiler version that is greater than or equal to `0.8.15` and less than `0.9.0`. It is recommended that contracts should be deployed with the same compiler version and flags used during development and testing. Locking the pragma helps to ensure that contracts do not accidentally get deployed using another pragma. For example, an outdated pragma version might introduce bugs that affect the contract system negatively.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

Lock the pragma version of all files to the same version used during development and testing.

Remediation Comment

SOLVED: The **Reserve Money team** fixed this finding in commit `cd73e8e` by locking the Solidity version in use as recommended.

Remediation Hash

https://github.com/rezervemoney/code/commit/cd73e8e986e9489ea8c848dc2e2a7d1037556c5d

7.9 USE OF REVERT STRINGS INSTEAD OF CUSTOM ERRORS

// INFORMATIONAL

Description

In Solidity smart contract development, replacing hard-coded revert message strings with the `Error()` syntax is an optimization strategy that can significantly reduce gas costs. Hard-coded strings, stored on the blockchain, increase the size and cost of deploying and executing contracts.

The `Error()` syntax allows for the definition of reusable, parameterized custom errors, leading to a more efficient use of storage and reduced gas consumption. This approach not only optimizes gas usage during deployment and interaction with the contract but also enhances code maintainability and readability by providing clearer, context-specific error information.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

It is recommended to replace hard-coded revert strings in require statements for custom errors, which can be done following the logic below:

1. Standard require statement (to be replaced):

```
require(condition, "Condition not met");
```

2. Declare the error definition to state:

```
error ConditionNotMet();
```

3. As currently is not possible to use custom errors in combination with require statements, the standard syntax is:

```
if (!condition) revert ConditionNotMet();
```

More information about this topic in the official Solidity documentation.

Remediation Comment

ACKNOWLEDGED: The **Reserve Money team** acknowledged this finding.

7.10 STYLE GUIDE OPTIMIZATIONS

// INFORMATIONAL

Description

The project codebase contains several stylistic inconsistencies and deviations from Solidity best practices, which, while not directly impacting functionality, reduce code readability, maintainability, and adherence to standard conventions. Addressing these inconsistencies can enhance the clarity and professionalism of the code.

- **Use of Whole File Imports Instead of Named Imports:** Full file imports are used, which may include unnecessary code and reduce clarity.
- **Use of `public` Where `external` Could Be Used:** functions like `initialize()`, `backingRatioE18()` or `calculateReserves()` are declared as `public` even though it could be declared as `external` to potentially save gas and adhere to best practices.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

Consider implementing the style improvements highlighted above to enhance the readability and consistency of the project.

Remediation Comment

ACKNOWLEDGED: The **Reserve Money team** acknowledged this finding.

7.11 CONSIDER USING NAMED MAPPINGS

// INFORMATIONAL

Description

The project accepts using a Solidity compiler version greater than `0.8.17`, which supports named mappings. Using named mappings can improve the readability and maintainability of the code by making the purpose of each mapping clearer. This practice helps developers and auditors understand the mappings' intent more easily.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

Consider refactoring the mappings to use named arguments, which will enhance code readability and make the purpose of each mapping more explicit.

For example, instead of declaring:

```
mapping(address => bool) public enabledTokens;
```

It could be declared as:

```
mapping(address token => bool enabled) public enabledTokens;
```

Remediation Comment

SOLVED: The **Reserve Money team** fixed this finding in commit `4c4d2c9` by removing the mappings of the contract.

Remediation Hash

https://github.com/rezervemoney/code/commit/4c4d2c9d57875b28a7e9abe545bde88497216fe3

7.12 UNSAFE ERC20 OPERATION IN USE

// INFORMATIONAL

Description

The `AppTreasury` contract contains an unsafe ERC20 operation that does not use the OpenZeppelin's `SafeERC20` library. This operation may fail silently with non-compliant tokens that return false instead of reverting on failure, or tokens like USDT that require resetting allowances to zero before updating them.

```
81  function withdraw(uint256 _amount, address _token)
82    external
83    override
84    nonReentrant
85    whenNotPaused
86    onlyReserveManager
87  {
88    require(enabledTokens[_token], "Treasury: not accepted");
89
90    uint256 value = tokenValueE18(_token, _amount);
91    app.transferFrom(msg.sender, address(this), value);
92    ...
```

Note: this instance was only reported as informational because the only unsafe ERC20 function detected was used to interact with the app (RZR) token.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

Consider using the OpenZeppelin's `SafeERC20` library consistently throughout the codebase.

Remediation Comment

SOLVED: The **Reserve Money team** fixed this finding in commit `cb40ced` by using the OpenZeppelin's `SafeERC20` library consistently as recommended.

Remediation Hash

https://github.com/rezervemoney/code/commit/cb40ced1158c1c410582e446244ea64200fd25a2

7.13 CACHE ARRAY LENGTH OUTSIDE OF LOOP

// INFORMATIONAL

Description

When the length of an array is not cached outside of a loop, the Solidity compiler reads the length of the array during each iteration. For **storage** arrays, this results in an extra `sload` operation (100 additional gas for each iteration except the first). For **memory** arrays, this results in an extra `mload` operation (3 additional gas for each iteration except the first).

Detecting loops that use the length member of a storage array in their loop condition without modifying it can reveal opportunities for optimization. See the following instances:

```
33  function enable(address _address) external onlyGovernor {
34    require(_address != address(0), "Zero address");
35
36    // RZR should not be enabled as a reserve; as this creates a circular dependency
37    require(_address != address(app), "RZR address");
38
39    // add token into tokens array if not already added
40    bool isAdded = false;
41    for (uint256 i = 0; i < tokens.length; i++) {
42      ...
```

And:

```
207  function calculateActualReserves() public view override returns (uint256 reserves) {
208    for (uint256 i = 0; i < tokens.length; i++) {
209      ...
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

Cache the length of the (storage and memory) arrays in a local variable outside the loop to optimize gas usage. This reduces the number of read operations required during each iteration of the loop. See the example fix below:

```
function enable(address _address) external onlyGovernor {
  require(_address != address(0), "Zero address");

  // RZR should not be enabled as a reserve; as this creates a circular dependency
  require(_address != address(app), "RZR address");

  // add token into tokens array if not already added
  bool isAdded = false;
  uint256 tokensLength = tokens.length;
  for (uint256 i = 0; i < tokens.length; i++) {
    ...
```

And:

```
function calculateActualReserves() public view override returns (uint256 reserves) {
  uint256 tokensLength = tokens.length;
  for (uint256 i = 0; i < tokensLength; i++) {
    ...
```

Remediation Comment

ACKNOWLEDGED: The **Reserve Money team** acknowledged this finding.

7.14 INCONSISTENT ERROR MESSAGES

// INFORMATIONAL

Description

Both `deposit()` and `withdraw()` guard against unapproved tokens but use different revert strings for the same condition:

```
function deposit(uint256 _amount, address _token, uint256 _profit)
  ...
  require(enabledTokens[_token], "Treasury: invalid token");
```

And:

```
function withdraw(uint256 _amount, address _token)
  ...
  require(enabledTokens[_token], "Treasury: not accepted");
```

This inconsistency can confuse integrators and complicate error handling in UIs and scripts, as they must account for multiple messages for the same failure case.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

Unify the revert reason (or migrate to a single custom error) so that both functions use an identical message. This ensures consistent on-chain semantics and simplifies off-chain error handling.

Remediation Comment

ACKNOWLEDGED: The **Reserve Money team** acknowledged this finding.

8. AUTOMATED TESTING

Static Analysis Report

Description

Halborn used automated testing techniques to enhance the coverage of certain areas of the smart contracts in scope. Among the tools used was Slither, a Solidity static analysis framework. After Halborn verified the smart contracts in the repository and was able to compile them correctly into their abis and binary format, Slither was run against the contracts. This tool can statically verify mathematical relationships between Solidity variables to detect invalid or inconsistent usage of the contracts' APIs across the entire code-base.

All issues identified by Slither were proved to be false positives or have been added to the issue list in this report.

Output

```
INFO:Detectors:
AppTreasury.withdraw(uint256,address) (contracts/AppTreasury.sol#81-98) ignores return value by app.transferFrom(msg.sender,address(this),value) (contracts/AppTreasury.sol#91)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#unchecked-transfer
INFO:Detectors:
AppTreasury.calculateActualReserves() (contracts/AppTreasury.sol#207-215) has external calls inside a loop: balance = IERC20(tokens[i]).balanceOf(address(this)) (contracts/AppT
reasury.sol#210)
        Calls stack containing the loop:
                AppTreasury.initialize(address,address,address)
                AppTreasury._updateReserves()
AppTreasury.tokenValueE18(address,uint256) (contracts/AppTreasury.sol#176-178) has external calls inside a loop: value_ = dreOracle.getPriceInAppForAmount(_token,_amount) (cont
racts/AppTreasury.sol#177)
        Calls stack containing the loop:
                AppTreasury.initialize(address,address,address)
                AppTreasury._updateReserves()
                AppTreasury.calculateActualReserves()
AppTreasury.calculateActualReserves() (contracts/AppTreasury.sol#207-215) has external calls inside a loop: balance = IERC20(tokens[i]).balanceOf(address(this)) (contracts/AppT
reasury.sol#210)
        Calls stack containing the loop:
                AppTreasury.setCreditReserves(uint256)
```

Halborn strongly recommends conducting a follow-up assessment of the project either within six months or immediately following any material changes to the codebase, whichever comes first. This approach is crucial for maintaining the project's integrity and addressing potential vulnerabilities introduced by code modifications.