Origin Dollar Audit

OCTOBER 25, 2021 | IN SECURITY AUDITS | BY OPENZEPPELIN SECURITY

The Origin Protocol team asked us to review and audit their Origin Dollar smart contracts. We looked at the code and now publish our results.

Scope

We audited commit `bf4ff28d5944ecc277e66294fd2c702fee5cd58b` of the Origin-Dollar repository. The scope includes all files in the following directories within `contracts/contracts/`, except specified exclusions:

- `buyback/`
- `flipper/`
- `liquidity/`
- `oracle/`, except `ChainlinkOracle.sol` and `MixOracle.sol`
- `strategies/`
- `token/`
- `utils/`
- `vault/`

In addition, imported interfaces were in scope. All other project files and directories (including tests), along with external dependencies and projects, were excluded from the scope of this audit. External code and contract dependencies were assumed to work as documented.

System overview

Origin Dollar (symbol: OUSD) is an ERC-20 compliant stablecoin backed by other stablecoins. The project is currently live, with OUSD deployed at 0x2A8e1E676Ec238d8A992307B495b45B3fEAa5e86 and the vault (for user interaction) deployed at 0xE75D77B1865Ae93c7eaa3040B038D7aA7BC02F70. Additional project addresses can be found on their registry.

OUSD stays fully collateralized and maintains a 1:1 peg to USD (approximated by the underlying DAI, USDC, and USDT) by taking favorable exchange rates when converting between OUSD and DAI, USDC, USDT via minting or redeeming, and investing that collateral to earn yield. The yield earned by the project is distributed to OUSD owners and a trustee address in the form of increased OUSD supply.

Users can mint OUSD by depositing either DAI, USDC, or USDT, which goes into the project's *vault*. The amount of OUSD minted depends on the current price of DAI, USDC, or USDT, as reported by Chainlink, in USD terms. The Origin Dollar project caps the value of DAI, USDC,and USDT at 1 USD, such that no more than 1 OUSD can be minted in exchange for 1 DAI, USDC, or USDT.

The project invests the DAI, USDC, and USDT into either Aave, Compound, or Curve to earn yield via the *strategy* contracts. Each underlying token (either DAI, USDC, and USDT) has a default strategy contract to which it is allocated. The project balances the underlying funds between the vault and strategy contracts, using a buffer, such that the vault remains liquid while most funds continue to earn yield.

Due to safety considerations, only externally owned accounts earn yield by default. However, contract addresses can opt-in to rebasing to share in the yield earned by the strategies, which will periodically increase the number of OUSD owned. The rebase process can also distribute a portion of the yield to the trustee. Currently, the trustee is a contract that implements an OGN buyback program: the OUSD is traded for OGN tokens through Uniswap to create buy pressure.

Users can redeem OUSD from the vault, receiving a pro-rata share of the value of DAI, USDC, and USDT in proportions equal to those held by the project. The price of the stablecoins is taken into account at redemption such that their value is capped at no more than 1 OUSD. Redeeming OUSD through the vault will incur a fee set by governance. The purpose is to incentivize long-term deposits to encourage stability.

In addition to these core contracts, we also reviewed some utility contracts that support auxiliary behaviors:

- There is a Flipper contract that is provided to users as a way to swap in and out of OUSD cheaply for any of DAI, USDC, or USDT at a fixed 1:1 rate. This contract will be used as an alternative way to route user transactions originating from the web app. This contract may become illiquid on one side (e.g., contain 0 OUSD balance), and thus sometimes provides limited swap routes.
- OUSD is governable and upgradeable. The Origin team is currently planning an upgrade to increase the resolution of rebasing calculations from 18 decimals to 27 decimals. The OUSD token itself will still retain 18 decimals of precision and user balances should not change.
- There is a mechanism to reward users that provide OUSD liquidity to third-party systems. The Origin team intends to run *campaigns* to distribute a specified number of OGN tokens amongst all users that lock approved LP tokens, thereby proving they contributed (or incentivized) liquidity.

Privileged roles

There are a variety of roles to manage access control, primarily used on the critical vault and strategy functions. There is a governor role, currently assumed by the 5-of-8 multisig at 0xbe2AB3d3d8F6a32b96414ebbd865dBD276d3d899, which is the administrator of the governor contract at 0x830622bdd79cc677ee6594e20bbda5b26568b781 . There is also a strategist role, currently assumed by the 2-of-9 multisig at 0xF14BBdf064E3F67f51cd9BD646aE3716aD938FDC. In addition, many functions are restricted to only being called by the vault.

There is overlap between these roles' capabilities. The most notable functions callable by either the governor, strategist, or

vault are provided below.

Notable functions callable by the governor address:

- `queue`, `cancel`, and `upgradeTo` for upgrade proposals
- `setStrategistAddr`, `approveStrategy`, `removeStrategy`, and `setAssetsDefaultStrategy`
- `setPriceProvider` and `setUniswapAddress`
- `setTrusteeAddr`
- `setRedeemFeeBps`
- `setVaultBuffer` used to balance funds between the vault and strategies
- `transferTokens` from the buyback contract to the governor address
- `withdrawAll` from the flipper contract to the governor address
- `setAutoAllocateThreshold` and `setRebaseThreshold` to handle automatic `allocate` and `rebase` while minting
- `setRewardTokenAddress`, `setRewardLiquidationThreshold`, and `harvest` to collect reward tokens from yield platforms
- functions to pause/unpause deposits and rebasing
- `withdrawAllFromStrategy` and `withdrawAllFromStrategies` to withdraw all assets from any or all strategies and send them to the vault
- `startCampaign` and `stopCampaign` for liquidity reward programs

Notable functions callable by the strategist address:

- `setVaultBuffer` used to balance funds between the vault and strategies
- `pauseRebase`, `pauseCapital`, and `unpauseCapital` to stop deposits
- `withdrawAllFromStrategy` and `withdrawAllFromStrategies` to withdraw all assets from any or all strategy and send them to the vault
- `harvest` to collect reward tokens from yield platforms and swap them for stablecoins

Notable functions callable by the vault address:

- `mint`, `burn`, and `changeSupply` of OUSD
- `swap` OUSD for OGN in the buyback contract
- `harvest` to collect reward tokens from yield platforms and `collectRewardToken` to transfer them to the vault
- `deposit`, `depositAll` and `withdraw`, into or out of strategies

Security model and trust assumptions

OUSD is governable and upgradeable. The governance system can be used to change many critical factors, as well as the pricing oracle, giving governance ultimate powers. Since the system is already deployed, we considered the current value of critical factors when assessing findings. We assume governance will act in the protocol's best interest, would not approve malicious or ill-advised upgrades, and maintains sensible values for sensitive variables.

The project also has many integrations, thus, it relies on trusting numerous systems to act in a well-defined manner. For example, the system relies on trusting Chainlink for accurate and available pricing, as well as trusting that Uniswap, Aave, Compound, and Curve will work as intended for swapping, depositing, withdrawing, and other investing functions. External code and contract dependencies were assumed to work as documented.

In addition, OUSD is fully backed by DAI, USDC, and USDT. Thus, the value of OUSD depends on the resilience of these underlying stablecoins. If, for example, DAI were to lose it's peg from USD and never return, this would cause economic

issues for OUSD, despite the automated mitigations. In such a case, an upgrade may be necessary to change OUSD token dynamics to maintain a 1:1 peg with USD.

Here we present our findings.

Update

Most of the following issues have been either fixed, partially fixed, or acknowledged by the Origin Team. Our analysis of the mitigations is limited to the specific changes made to cover the issues, and disregards all other unrelated changes in the pull requests and in the codebase.

Critical severity

[C01] Attacker can steal a portion of the reward tokens and accrued yield

The `_allocate` function in the `VaultCore`:

- harvests reward tokens and swaps them on Uniswap in exchange for USDT, using WETH as an intermediary.
- calls the `swap` function on the `Buyback` contract, which swaps all of its OUSD on Uniswap in exchange for OGN.

In both cases, the protocol uses the `swapExactTokensForTokens` function on Uniswap's `UniswapV2Router02` contract and sets the minimum number of tokens to receive to zero. In other words, there is no slippage protection, which makes the trades vulnerable to price manipulation.

As an example, consider a scenario where the vault is on the verge of collecting and liquidating its Compound reward tokens, COMP. An attacker could:

1. Flash-borrow a huge amount of COMP and sell it in the COMP/WETH Uniswap pool to significantly lower the COMP price
2. Call the `mint` or `mintMultiple` functions to trigger an allocation and swap the harvested COMP tokens. Due to the price movement in step 1, the contract will receive less than the market value. This also lowers the COMP price further.
3. Sell the WETH back in the Uniswap pool to recover the COMP, which will be even cheaper due to step 2.
4. Repay the loan. Since the attacker buys COMP at a cheaper rate than it was initially sold, they profit the difference.

Similarly, the attacker can perform a flash-loan to borrow OUSD and manipulate the OUSD/OGN price in Uniswap (by moving the OUSD/USDT and/or the OGN/WETH price), to extract some of the value of the OUSD swapped by the `Buyback` contract. In the worst-case scenario, the attacker would perform this same attack in the same transaction for all the reward tokens collected from all the strategies, and for the OUSD swapped by the `Buyback` contract.

Note that the profit that the attacker can make from this attack depends on the amount of money deposited in the strategies, since the amount of reward tokens collected and the yield accrued are proportional to this amount. Consequently, the feasibility of this attack increases as the platform's investments grow.

Additionally, it is worth mentioning that this same attack can be performed without using a flash-loan, by sandwiching a call to the `allocate` function, which can be called by anyone, or by sandwiching a call to any of the versions of the `harvest` function in the `VaultAdmin` contract, which can be called by the governor or the strategist.

Consider adding slippage protection to all calls to the `swapExactTokensForTokens` function from the `UniswapV2Router02` contract.

Update: *Fixed in PR#624 and PR#724. However, the `BuybackConstructor` still references the Uniswap V2 router.*

High severity

[H01] Resolution upgrade inconsistency

The OUSD token achieves its rebasing functionality by tracking credit balances and scaling them by a conversion factor to retrieve the corresponding OUSD token balances. The `OUSDResolutionUpgrade` contract is designed as a temporary logic contract that replaces the token functionality with mechanisms to increase the precision of the conversion factors. In particular, there is a function to update the global parameters and a separate function to upgrade the individual user accounts in batches.

To avoid upgrading the same account multiple times, an upgrade flag is set for each account. Similarly, the upgrade flag is set for the zero address to indicate that the global parameters have been updated. There is no access control on either of these functions. This means that an attacker can include the zero address in a batch of account upgrades, which will set its flag and prevent anyone from upgrading the global state. This could produce an inconsistent state where a subset of the accounts use the new resolution, while the global parameters remain unchanged.

Consider restricting the `upgradeAccounts` function to non-zero account.

Update: *Fixed in commit 95e8c90.*

[H02] Valid redemptions may fail

The `_calculateRedeemOutputs` function in the `VaultCore` contract calculates the amount of each asset to be redeemed based on the total amount held by the vault and strategies. For the latter, this amount is calculated by summing the balance of each asset in each strategy, without considering whether this strategy is the asset's default strategy.

The `_redeem` function uses the calculated redeem outputs, then iterates through the approved assets to try to withdraw the total asset balance (across all strategies) from *only* the asset's default strategy, unless the vault already has a sufficient balance for the redemption.

Since the output ratios include balances held in non-default strategies, and `_redeem` only withdraws from the default strategy, this mismatch may cause the redemption process to fail in some scenarios where the protocol has enough liquidity. This can happen, for instance, when setting a new default strategy for a given asset without reallocating a sufficient amount from the old default strategy to the new default strategy.

Consider using the asset balance in all strategies to pay back the caller, instead of just the default strategy. Alternatively, consider ensuring that each asset is only invested through its default strategy.

Update: *Not fixed. The Origin team states:*

> We'll keep this the way it is. Some yield earning protocols are inherently attackable when users can force OUSD to move funds into and out of them, either from entrance/withdrawal fees or economic attacks. In order to be able to use these, we have to have funds that can't be deposited to or withdraw under direct user control. The allocations into and out of non-default strategies is currently handled by the strategist role, and we are planning on transitioning this funds allocation to community governance.

[H03] Manipulable rewards calculations

When a user deposits LP tokens into the `LiquidityReward` contract, they are entitled to a proportional share of subsequently released rewards, which is tracked as a credit of the equivalent share of all accumulated rewards and a debt of the corresponding rewards that were released before the deposit, which the user should not receive. The aggregate effect is calculated using the contract's LP token balance and the `totalRewardDebt`.

However, if a user simply transfers LP tokens to the contract directly, their tokens will be captured in the contract's balance but not the `totalRewardDebt`. In such a scenario:

- `startCampaign` will overestimate the total pending rewards, which will increase the OGN necessary to start the campaign.
- The `totalOutstandingRewards` function will return an overestimate for the same reason.

Consider tracking the amount of LP token deposits so valid deposits can be distinguished from the contract balance.

Update: *Fixed in PR#688.*

[H04] Incorrect slippage in Curve 3Pool strategy

When withdrawing funds from the `ThreePoolStrategy`, LP tokens are exchanged for the underlying asset. To control slippage, a minimum asset amount to receive is specified.

However, the actual parameter does not match the intended usage. Instead, it is calculated as a fraction of the burned LP tokens, scaled so it has the same precision as the asset to withdraw. This is not a meaningful value, and when interpreted as an asset quantity, it may be more than the burned LP tokens are worth. In this scenario, the liquidity removal will fail unexpectedly.

To avoid this scenario, consider setting this parameter to `_amount`. Given the way the number of tokens to burn was calculated, they will always exchange for `_amount` of assets at least. This would match the current behavior while avoiding the possibility of an unnecessary failure to withdraw.

Unfortunately, this suggestion does not protect against a front-running attack or sandwich attack, where the instantaneous state of the Curve protocol differs significantly from market equilibrium. To mitigate this, the `ThreePoolStrategy` contract would need the fair market rate of LP tokens denominated in the asset to withdraw.

Update: *Partially fixed by PR#716. The `withdraw` function will now remove at least `_amount` from Curve's 3Pool. Note, however, that the `withdraw` function still does not protect against front-running or sandwich attacks.*

[H05] AAVE inconsistency

The `AaveStrategy` contract uses the version 2 interface in anticipation of a future reconfiguration of the investment strategies. However, we identified two inconsistencies:

- the strategy attempts to grant an allowance to the non-existent Lending Pool Core, instead of the Lending Pool, which would make the strategy unusable and prevent OUSD token mints whenever the strategy is in use.
- the strategy uses the outdated `redeem` function in the `withdrawAll` function, which is called when the strategy is removed.

When this was raised with the Origin team they indicated that they had already identified and addressed the first inconsistency in a subsequent commit. Consider using the new interface when withdrawing all tokens.

Update: *Fixed in commit 650913e.*

Medium severity

[M01] Mint after balance check

The `collectRewardToken` function of the `ThreePoolStrategy` contract mints any outstanding reward tokens and transfers its reward balance to the vault. However, the outstanding tokens are minted after the reward balance is retrieved and the event is emitted. This means that the newly minted tokens are not sent to the vault, and will only be transferred in a subsequent call

to `collectRewardToken`.

Consider minting the new reward tokens first to ensure they are included in the transfer.

Update: *Fixed in PR#640.*

[M02] Not enforcing a default strategy for new assets

The `supportAsset` function allows the governor to add a new asset to the `VaultCore` contract, but it does not enforce a default strategy for it by calling the `setAssetDefaultStrategy`.

This can potentially cause a misbehavior when allocating assets from the vault to the strategies. The new asset can still be deposited through the `mint` or `mintMultiple` functions, and it will contribute to the utilization of the vault buffer, but it will not be deposited to any strategy. This skews the investment ratio so a higher percentage of *all the other assets* will be moved to the strategies, draining the buffer that should be used for future redemptions.

Consider enforcing configuration of the asset's default strategy in the `supportAsset` function.

Update: *Not fixed. The Origin team states:*

> We'll keep this the way it is. This won't lose any funds if it's not set. The vault still operates without a default strategy (both places it is used, allocating and redeeming check if this is set and skip it if it is not). It is possible that there may be temporary times in DeFi when the Origin Dollar doesn't trust any lending protocol/strategy, and goes to purely holding assets to increase stability until things settle down.

[M03] Not checking asset balance in strategy before removal

The `removePToken` function from the `InitializableAbstractStrategy` lets the governor remove an asset from the strategy, by:

- Removing the asset address from the `assetsMapped` array (e.g., DAI, USDT, USDC addresses).
- Setting the `assetToPToken` mapping for that particular asset address to `address(0)`.

However, there are no checks for whether the strategy still invests the asset in the underlying platform before removing it. If the asset is removed, this invested amount will be disregarded when checking the total amount of assets held by a strategy, which is used to perform allocations, or when calculating the redeem outputs, which is used to perform redemptions. Additionally, the invested amount will be disregarded in the `withdrawAll` functions defined on each child contract.

Consider either reverting if the strategy still invests the asset, or sending the remaining balance to the vault contract.

Update: *Not fixed. The Origin team states:*

> We'll keep this as it. We can recover funds if we need to by re-adding the strategy. By not checking amounts etc, it gives us a way to remove a strategy that is broken in some way, and prevents some other project's DOS from being our DOS for long.

[M04] Trapped Liquidity Rewards

Whenever a liquidity reward campaign is initiated, the `LiquidityReward` contract ensures the contract is preloaded with enough reward tokens to execute the campaign. However, some of these rewards would not be distributed if the campaign is stopped. In this scenario, the excess reward tokens cannot be retrieved from the contract. It would be possible to start a new campaign, but then the funds would be distributed to the existing depositors, which may not be desired (and likely undermines the reason for stopping the campaign). Consider introducing a mechanism to retrieve reward tokens that are not intended for distribution.

Update: *Fixed in PR#688.*

[M05] Excessive Curve 3Pool withdrawal

When withdrawing funds from the `ThreePoolStrategy`, the number of LP tokens to burn is determined by retrieving the asset value of all LP tokens, and then scaling down linearly to the desired withdrawal amount. However, not all LP tokens are valued equally: as the size of the withdrawal increases, the value of each LP token should decrease. This means the withdrawal will retrieve too many tokens.

To account for this, any excess tokens are sent to the vault. The comments and variables names suggest that the excess amount would be negligible. However, since most use cases involve withdrawing a small fraction of all the assets invested by this strategy, and the discrepancy increases as the fraction decreases, it could be significant. It is worth noting that excess funds that are sent to the vault do not automatically trigger a reallocation if they exceed the internal liquidity buffer.

To avoid excess withdrawals, consider using the `calc_token_amount` function to determine the number of LP tokens to burn or the `remove_liquidity_imbalance` function to withdraw a specific amount of asset tokens.

Update: *Partially fixed in PR#718. Although `remove_liquidity_imbalance` is now used to avoid excess withdrawal from Curve's 3Pool, the strategy contract still withdraws, at most, the maximum amount needed from the Gauge. In addition, the strict in inequality in this require statement makes it impossible to withdraw the max amount of `pTokens`.*

Low severity

[L01] Unhandled token transfer fees

The `mint` and `mintMultiple` functions of the `VaultCore` contract implicitly assume the token deposit will transfer the specified number of tokens. However, the system supports the `USDT` token, which includes the possibility of transfer fees. Consider minting OUSD tokens based on a measurement of the amount of received tokens.

Update: *Not fixed. The Origin team states:*

> We will be leaving this as is. It is our current belief that enabling transactions fees on Tether would destroy it, given that its entire success is based on being the lowest friction stable medium of exchange. In the event that Tether enables fees, we'll adjust at that time.

[L02] Cannot redeem from both accounts simultaneously

When redeeming OUSD tokens, the `VaultCore` contract will retrieve all the tokens from either its own buffer or the default strategy. However, if neither the buffer nor the strategy has sufficient balance individually, the transfer will fail, even if the combined balance is large enough and the redemption is valid. This could occur for large redemptions or unbalanced stablecoin reserves, and gets worse when the `vaultBuffer` is high.

Consider withdrawing from both the buffer and the strategies when required.

Update: *Not fixed. The Origin team states:*

> We are going to keep this the way it for now. We would rather have less complexity in the code here. The buffer in OUSD is a gas optimization to keep gas usage from being extremely expensive during non-huge mints and redeems, and only needs to keep a very small portion of the total vault value in the buffer. Right now, it's targeting a half a percent of the assets, and as the asset amount goes up further, this percentage will be lowered more. While there remains a reasonable amount of assets in OUSD, this shouldn't be a problem. In the case that almost all assets are being withdrawn, it should still be possible to get funds out, it just may take carefully chosen withdrawal amounts.

[L03] Residual token allowance

There are multiple places in the codebase where addresses may retain excess token allowances:

- the `setUniswapAddress` function of the `BuyBack` contract grants an OUSD allowance to the new address, but does not revoke the allowance of the old address, if it exists.
- the `_abstractSetPToken` function and `safeApproveAllTokens` functions of all three strategies grants an allowance to the investment platform to spend assets on behalf of the strategy, but the `removePToken` does not revoke this allowance.
- the `_harvest` function of the `VaultAdmin` contract grants an allowance for Uniswap to spend reward tokens, but changing the Uniswap address, changing the reward token, or removing the strategy does not revoke this allowance.

Consider revoking token allowances when they are no longer required.

Update: *Acknowledged and retained. The Origin team states:*

> The strategy and buyback contracts are designed to be short-lived if necessary. The major change of switching underlying systems would probably be accompanied by a clean proxy and a clean slate of storage.
>
> The vault though is designed to last for the long term. However the only approvals it currently does are for the exact amount of the swap being made. This means that there should be no left over approvals afterwords that need to be revoked.

[L04] Rounding down in OUSD calculations

When transferring tokens, the `OUSD` contract translates the token amount to the equivalent number of credits. However, the value is rounded down in both directions. As a general principle, to minimize the attack surface, rounding errors should be in favor of the protocol. Consider using the `mulTruncateCeil` function to calculate `creditsDeducted`.

Similarly, consider using the `mulTruncateCeil` function when calculating the number of credits to burn. This would also remove the need to check for rounding errors when burning the whole balance.

Update: *Not fixed. The Origin team states:*

> We are going to keep this as is for now. While rounding in the protocol's favor is generally good, any rounding against the users' transfer, mint, and burn amounts gets quite painful for users. Users expect that if they send someone X, the recipient will get X, and that if they want to keep Y, and they send their balance minus Y, that they will still have Y. The amounts involved are trillionths of a trillionth of a dollar, and OUSD has a flow of yield coming into it.

[L05] Not leaving a storage gap in upgradeable contracts on multi-level inheritance

The `CompoundStrategy`, `AaveStrategy`, and `ThreePoolStrategy` contracts inherit from `InitializableAbstractStrategy`, and are all meant to be upgradeable, which means that the governance can add new functions and/or storage variables to them. However, if a new storage variable is added to the `InitializableAbstractStrategy` contract, it will overwrite the storage of the children contracts and therefore will make the storage layout incompatible.

To allow additions of new state variables without compromising the storage compatibility with existing deployments, consider leaving a storage gap at the end of each newly deployed upgradeable contract that has children contracts. Note that this should only be performed in newly deployed contracts, *and not between upgrades*.

Update: *Fixed in PR#713.*

[L06] Rewards can overflow buffer

When allocating funds, the `VaultCore` contract first ensures its internal buffer is full and then determines the excess amount to invest. After investing this amount for each asset, it harvests any rewards from its previous investments and converts them to USDT.

This implies that the reward balances, which can be arbitrarily large, remain in the vault and exceed the size of the buffer. This undermines the purpose of the buffer and is an unexpected behavior. Consider harvesting the rewards before determining and allocating the available funds.

Update: *Fixed in PR#690. The reward tokens are now harvested before allocating assets to the strategies.*

[L07] Outdated Solidity version in use

An outdated Solidity version, `0.5.11`, is currently in use. As Solidity is now under a fast release cycle, consider using the latest compiler version at the time of future deployments and upgrades (presently `0.8.7`) to incorporate the latest bug fixes. Note that although Solidity uses semantic versioning, it may introduce breaking changes between its minor versions, which should be taken into consideration.

Update: *Fixed in PR#738.*

[L08] Simplify redeem outputs calculation

The calculation to split a value into a fair distribution of coins is unnecessarily complicated. In particular, the total balance of all stablecoins is redundant and can be ignored (or set to 1), which makes the calculation easier to reason about, as follows:

- the `ratio` parameter would become the product of the balance and the price (i.e., the value of the asset held in the vault).
- the `totalOutputRatio` parameter could be renamed to `totalValue`, since it represents the value held by the vault.
- the `factor` parameter now corresponds to the fraction of the vault that is being redeemed.
- the `outputs` parameter has the same value as before, but now it can be directly interpreted as the same fraction of each asset balance.

Consider removing the `totalBalance` parameter from the calculation.

Update: *Acknowledged. The Origin team states:*

> We'll keep this the way it is. While it is correct that calculating the total balance of all stablecoins is not a requirement to calculate the outputs, returning that number from the function does provide a considerable gas optimization for the redeem process. Getting the total balance is very expensive, since each strategy needs to check each stablecoin that it supports, and the strategy needs to check each place that it could have assets, and some of the strategies assets are not directly denominated in stables, which require further computation to get the exchange rate. Since we are doing all this work anyway in this output calculation function, we do just extra three adds to sum to a total value here and return it for use outside this function.

[L09] Mixing testing and production code

The codebase contains a few duplicated contracts with slightly different, security-relevant behaviors. Specifically:

- The `BuybackConstructor` contract is equivalent to the `Buyback` contract, except it allows the deployer to arbitrarily set the contract addresses. It also saves these values as contract variables, which causes the storage layout to differ between the contracts.

- The same observation applies to the `FlipperDev` and `Flipper` contracts.
- The `OracleRouterDev` contract is equivalent to the `OracleRouter` contract, except it allows anyone to arbitrarily set the oracle addresses.

In all cases, we expect the "Constructor" and the "Dev" contracts are intended for testing purposes only, which explains their increased flexibility. As a matter of good practice, consider documenting this in the function comments and moving them to a `test` directory to maintain clearer isolation between the testing and production code.

Update: *Acknowledged. The Origin team states:*

> The issue has an easy solution with Solidity's immutable feature, so that will be implemented once the Solidity version is upgraded.

[L10] Inconsistent interfaces

The codebase defines an `IStrategy` interface that is not inherited by any of the strategies. Similarly, the `IVault` interface is not inherited by the vault. This has introduced some inconsistencies:

- The `IVault` interface declares `DepositsPaused` and `DepositsUnpaused` events that are never used
- It also declares a non-existent `checkBalance` function with no parameters.
- The `reallocate` function is declared in the `VaultCore.sol` section, but it is defined in `VaultAdmin.sol`

Consider inheriting the interfaces whenever they are implemented and correcting these discrepancies.

Moreover, the `IUniswapV2Router` interface only defines the subset of the interface that is needed in the codebase. Nevertheless, it includes the unused `addLiquidity` function. Consider removing it.

Update: *Fixed in PR#689.*

[L11] Missing validation

There are currently some unvalidated assumptions in the codebase. For example:

- The `VaultCore` contract checks if the trustee is defined before sending it fees, but assumes it is defined and has a `swap` function when triggering the buyback mechanism. If this assumption does not hold, the contract will be unable allocate funds.
- The `ThreePoolStrategy` contract makes assumptions that `assetsMapped` is of length 3, but there's nothing in the `_initialize` function that enforces that. The `deposit` and `depositAll` functions would revert if the `assetsMapped` were ever more than 3 elements long.
- The `startCampaign` function from the `LiquidityReward` contract does not validate that `_numBlocks` is not zero. This could emit confusing events and update values such as `startBlock`, `endBlock`, and `pool.lastRewardBlock` without reason.

Consider adding validation in these and all other places where assumptions are currently unchecked to reduce the chance of errors when interacting with and refactoring the contracts.

Update: *Fixed in PR#632, PR#688, and PR#715.*

[L12] Function selectors between `VaultCore` and `VaultAdmin` could collide

Instead of inheriting from the `VaultAdmin` to access to its functionality, the `VaultCore` contract forwards calls using `delegatecall` in the `fallback` function. This means that if a function is called that does not exist in `VaultCore`, the `fallback`

function will forward this call to the `VaultAdmin` contract, using the context of the former.

The issue is that function selectors between the `VaultCore` contract and the `VaultAdmin` contract are not being checked for possible collision during deployment. This means that, if in the future (e.g., through a governance upgrade) an external or public function is defined in the `VaultAdmin` that has the same selector as one of the functions in the `VaultCore` contract, the `VaultAdmin` function will be impossible to call.

Consider checking that there are no collisions between the functions defined in the `VaultCore` and the `VaultAdmin` contracts within the deployment scripts.

Update: *Acknowledged. The Origin team states:*

> We are planning on writing a script to check this.

[L13] Lack of event emission after sensitive actions

The following functions do not emit relevant events after executing sensitive actions.

- The `setRewardTokenAddress` function of the `InitializableAbstractStrategy` contract should emit a `RewardTokenAddressUpdated` event when updating the `rewardTokenAddress` variable.
- The `setRewardLiquidationThreshold` function of the `InitializableAbstractStrategy` should emit a `RewardLiquidationThresholdUpdated` event
- The `withdrawAll` function in the `CompoundStrategy` contract should emit the `Withdrawal` (or a new `WithdrawAll`) event as the `withdraw` function does.
- The `_allocate` function from the `VaultCore` contract should emit an `AssetsAllocated` event.

Consider emitting events after sensitive changes take place, to facilitate tracking and notify off-chain clients following the contracts' activity.

Update: *Partially fixed in PR#677. The `withdrawAll` function still does not emit events.*

[L14] Incomplete event emissions

When defining events with the sole purpose of showing a storage modification, it is a good practice to emit both the old value and the new value of the modified variable. *Some examples* are:

- The `UniswapUpdated` event in the `Buyback` and `BuybackConstructor` contract should emit both the old and the new uniswap addresses.
- The `PriceProviderUpdated`, `RedeemFeeUpdated`, `VaultBufferUpdated`, `AllocateThresholdUpdated`, `RebaseThresholdUpdated`, `UniswapUpdated` events (among others) in the `VaultAdmin` contract should emit the old and new value that is being updated.

Additionally, the `Withdrawal` event in the `ThreePoolStrategy` contract should also emit the beneficiary address (i.e., who receives the assets).

Consider reviewing all the events that are being emitted throughout the codebase and checking that all sensitive variables are being emitted, to avoid hindering the task of off-chain services interested in these events.

[L15] Missing error messages in `require` statements

Some `require` statements are missing error messages, such as the following:

- the `transfer` calls in the `Flipper` contracts
- the require statements within the `upgradeGlobals` and `upgradeAccounts` functions of the `OUSDResolutionUpgrade` contract

To improve the code's readability and to help debugging issues that may arise, consider including specific and informative error messages in all `require` statements.

Update: *Fixed in PR#662.*

[L16] Lack of indexed parameters in events

Throughout the code, there are parameters in events that are not indexed. *Some examples* are:

- In the `VaultStorage` contract: `AssetDefaultStrategyUpdated`, `Mint`, `Redeem`, `YieldDistribution`, `TrusteeAddressChanged`, `StrategistUpdated`, `UniswapUpdated`, `PriceProviderUpdated`.
- The `asset` parameter in the `SkippedWithdrawal` event.
- The `recipient` parameter in the `RewardTokenCollected` event.

Consider indexing event parameters to avoid hindering the task of off-chain services searching and filtering for specific events.

Update: *Acknowledged and retained. The Origin team states:*

> Keeping as is to maintain backwards compatibility.

[L17] Not using `safeTransfer`

The `Flipper` contract allows users to exchange OUSD 1:1 for any of DAI, USDC, or USDT and vice versa as a low cost way to perform swaps. Although the `withdraw` functions use `safeTransfer`, none of the other `transfer` functions in the Flipper contracts use `safeTransfer`.

Consider always using `safeTransfer` as a best practice.

Update: *Acknowledged. The Origin team states:*

> The flipper contract uses hard-coded token addresses, and does not support adding tokens without deploying a new contract. For the swaps, each token has the correct interface for it, with USDT in particular using their own returnless transfers. We do use `safeTransfer` on methods that can operate on arbitrary tokens. We're going to keep this as is, since the focus on this contract is extremely low gas usage for small swaps.

[L18] Implicit casting

Implicit casting is used to convert the `price` from `int256` to `uint256`. This could overflow if the price were negative, but the require statements should revert the function in such case, as the result would be outside this range.

Although this cannot result in overflow because of the `require` statements, consider using SafeCast to safely convert between different integer types as a best practice.

Update: *Acknowledged and retained by the Origin team.*

[L19] Unnecessary empty constructors defined

The `Flipper` contract defines empty constructors with no parameters, which is not necessary and only hinders code

readability. According to the Solidity docs on constructors: "If there is no constructor, the contract will assume the default constructor, which is equivalent to `constructor() public {}`".

To favor simplicity, consider removing all empty constructors from the codebase.

Update: *Fixed in PR#646.*

Notes & Additional Information

[N01] Buyback could be rebasing

The `Buyback` contract can receive OUSD tokens whenever the token contract rebases and it disposes of them whenever funds are allocated in the vault and it has at least $1000. Therefore, it's possible for the contract to hold OUSD tokens during a rebase. Moreover, it does not perform any internal accounting to track its own OUSD balance, which makes it safe for rebasing. To maximize the effectiveness of the Buyback program, consider allowing the contract to opt-in to rebases.

Update: *Acknowledged and retained. The Origin team states:*

> We'll keep as is. By not being rebasing, this provides extra yield to user of OUSD.

[N02] Misleading comments

Throughout the code, we found several comments either misleading or inconsistent with the function implementation. *Some examples* are:

- The `withdrawAll` functions of the `Flipper` contracts claim to be equivalent to "pausing" the contract, but this is not true as anyone can still subsequently deposit into the contract to enable all functionality.
- The `deposit`, `_deposit` and `withdraw` functions of the `AaveStategy` contract provide a NatSpec comment for the return value, but the function does not return anything.
- The `changeSupply` function of the `OUSD` contract provides a NatSpec comment for the return value, but the function does not return anything.
- The `onlyVault` modifier mentions a non-existent *Savings Manager* contract
- The `transferTokens` function on the `Buyback` contract is missing its `@param` statements.
- The `feed` function on the `OracleRouter` contract is missing its `@return` statement.

Consider either revising or removing misleading comments to more accurately reflect function implementations.

Update: *Partially fixed in commit `b80180bb5c606ea47266f186d5d232de95f72e48`, where the misleading `@return` NatSpec comments were removed.*

[N03] Withdraw funds to recipient

The `Buyback` contract and `Flipper` contract each contain mechanisms for the governor to withdraw funds to its own address. However, this mixes roles and requires additional complexity to handle the received tokens. Consider specifying a `recipient` address, so the governance structure can allocate the funds without having to first take possession of them.

Update: *Not fixed. The Origin team state:*

> We are going to keep these as is. It makes it easier for humans to validate the admin transactions, and it's one less place to fat finger a number. It would save on gas, but these admin transfers are very rare. When we do need to make them, our governor timelock can queue and atomically run multiple actions, so it's easy enough to withdraw in one action and

> transfer after, all in the same transaction.

[N04] Unnecessary external calls

In the `depositAll` function of the `ThreePoolStrategy` contract, the `get_virtual_price` function is called against Curve's 3Pool contract for each iteration of a *for loop*.

Consider instead just calling `get_virtual_price` once before the loop to improve efficiency.

Update: *Fixed in PR#639.*

[N05] Unnecessary writes to `strategies` mapping

The `removeStrategy` function removes a strategy from the vault, withdrawing all invested assets and returning them to the vault. It updates the `strategyIndex` depending on whether the `_addr` parameter exists in `allStrategies`. Then, if the `strategyIndex` exists in `allStrategies`, the function pops the value from the `allStrategies` array, and withdraws all assets from the strategy. Afterwards, the function updates the struct in the `strategies` mapping for `_addr` to set `isSupported` to false, regardless of whether the `_addr` key exists in `strategies`.

Consider moving the `strategies` mapping update into the conditional block that checks whether the `_addr` exists in `allStrategies` to avoid unnecessary writes to storage.

Update: *Fixed in PR#705.*

[N06] Unconventional storage slots

The `Governable` contract uses a similar pattern to EIP-1967 to produce pseudo-random storage slot locations. As noted in the EIP, it is conventional to introduce a fixed offset to ensure there is no known hash pre-image. Nevertheless, the current locations are chosen securely and we do not recommend changing storage locations on live contracts, so we are noting this for informational purposes.

Update: *Acknowledged. In the words of the Origin team:*

> We are acknowledging the comment on this, and concur with you that we should continue to use the current values.

[N07] Incorrect function visibility

The `allocate`, `rebase`, and `redeem` functions are not called internally by the `VaultCore` contract. Consider setting the visibility to `external` instead of `public`.

Update: *Fixed in PR#641.*

[N08] Lack of explicit visibility in state variables

Throughout the codebase there are state variables and constants that are implicitly using the default visibility. *Some examples* are:

- In the `VaultStorage` contract: `assets`, `allAssets`, `strategies`, `allStrategies`, `OUSD`.
- In the `ThreePoolStrategy` contract: `crvGaugeAddress`, `crvMinterAddress`, `maxSlippage`.

To favor readability, consider explicitly declaring the visibility of all state variables and constants.

Update: *Partially fixed in PR#642.*

[N09] Unnecessary modifier defined in function

The `rebase` function from the `VaultCore` contract uses the `whenNotRebasePaused` modifier and calls the `_rebase` internal function, which *also* uses this modifier.

In the interest of simplicity and avoiding redundant validations, consider removing the modifier from the `rebase` function.

Update: *Fixed in PR#643.*

[N10] Using `now` instead of `block.timestamp`

The global variable `now` is used in a few places within the codebase, such as in the `Buyback` contract and in the `VaultAdmin` contract. This value could be misinterpreted and has since been deprecated in Solidity v0.7.0.
Consider instead using `block.timestamp` to reflect that the value is a property of the block and to future-proof the codebase for newer versions of Solidity.

Update: *Fixed in PR#714.*

[N11] Using require to revert

The `OracleRouter` contract uses a `require` statement that always fails. To better signal the code's intention, consider using a `revert` statement instead.

Update: *Fixed in PR#644.*

[N12] Unused variable

The `calculateRedeemOutputs` function defines a `totalValue` variable, but never uses it.

Consider removing unused variables.

Update: *Fixed in PR#638.*

[N13] Inconsistent coding style

The codebase does not follow a consistent style and it deviates from the recommended Solidity Style Guide. Some examples include:

- constants not using UPPER_CASE format
- contract should be preceded by 2 blank lines
- the order of functions does not always follow the recommended order of: constructor, fallback, external, public, internal, private

Taking into consideration how much value a consistent coding style adds to the project's readability, enforcing a standard coding style with help of linter tools such as Solhint is recommended.

Update: *Acknowledged and retained by the Origin team.*

[N14] Typographical errors

We have identified the following typographical errors in the codebase:

- "ICERC20" should be "IERC20"

- "approval approval" should be "approval"
- "liquidiity" should be "liquidity"
- "ot" should be "of"
- "jGeneric" should be "Generic"
- "a" should be "an"
- "9e38" should be "9e36"
- "Addresss" should be "Address"
- "form" should be "from"
- "the the" should be "to the"
- "optionaly" should be "optionally"
- "suppported" should be "supported"
- "_amount" should be "amount" in several comments in the `OUSD` contract
- "to" should be "from"

Consider correcting typographical errors in the codebase and using an IDE add-on to identify errors in the future.

Update: *Fixed in commit 192e012.*

Conclusions

1 critical and 5 high severity issues were found. Some changes were proposed to follow best practices and reduce the potential attack surface.

RELATED POSTS

SECURITY AUDITS

Freeverse Audit

Th

SECURITY AUDITS