Empty Set V2 Audit

OPENZEPPELIN SECURITY | APRIL 1, 2021

Security Audits

Introduction

After auditing the first version of the protocol, the Empty Set Squad team asked us to audit their second version of the protocol, namely `emptyset` (formerly named `dollar-continuous`).

The repository can be found here and we audited the `master` branch on commit `bf9753ef9cd5b17236036257f290e0d0b850a029`.

The scope of the audit has been limited to the contracts present in the `protocol/contracts/src` folder. These were:

```
common/IImplementation.sol
common/Implementation.sol
common/ProxyRoot.sol
governance/GovernerAlpha.sol
governance/Stake.sol
incentivizer/Incentivizer.sol
Interfaces.sol
lib/Decimal.sol
lib/TimeUtils.sol
migrator/Migrator.sol
oracle/Oracle.sol
registry/Registry.sol
reserve/ReserveComptroller.sol
```

```
reserve/ReserveSwapper.sol
reserve/ReserveVault.sol
stabilizer/StabilizerComptroller.sol
stabilizer/StabilizerImpl.sol
stabilizer/StabilizerState.sol
stabilizer/StabilizerToken.sol
token/Dollar.sol
vester/Vester.sol
```

The following contracts were left out from the scope:

```
governance/Timelock.sol
lib/LibEIP712.sol
lib/UniswapV2Library.sol
lib/UniswapV2OracleLibrary
token/Permittable.sol
vester/TokenVesting.sol
```

All other project's files and directories (including tests), along with external dependencies and projects, game theory, and incentive design, were also excluded from the scope of this audit.

Project overview

The new version of the protocol introduces several differences from the previous one. The new protocol creates a stablecoin called `ESD` which relies on different mechanisms to stabilize the `ESD` price around 1 `USDC`. The new protocol consists in 3 new tokens: `ESD`, which is represented by the `Dollar` contract, `ESDS`, which is represented by the `Stake` contract, and `sESD`, implemented by the `StabilizerToken`, all of 3 described below.

The project consists of the following parts:

Governance

successfully voted proposals and to allow users to eventually exit the protocol if they do not agree with the queued proposals. Similarly, the `Stake` contract provides the `ESDS` governance tokens to let users have vote powers, and choose a delegatee to vote on proposals.

The `GovernorAlpha` contract has a special role called the `guardian` which is set in the constructor and has special permissions to execute the following actions:

- Cancel any non-executed proposal via the `cancel` function.
- Bypass the governance mechanism to change the Timelock's admin at will (calling the `GovernorAlpha` contract `__queueSetTimelockPendingAdmin` and `__executeSetTimelockPendingAdmin` functions), meaning that the Governor's guardian can override any queued proposal that attempts to change the `Timelock` contract's admin (i.e., a proposal that calls the `setPendingAdmin` function), as the guardian can call `__queueSetTimelockPendingAdmin` with any eta.
- Make the `GovernorAlpha` contract accept administrative powers over the Timelock contract, calling the `__acceptAdmin` function.
- Abdicate, by calling the `__abdicate` function.

Registry

It is used as a unique source of information, almost all contracts query protocol contract addresses to the `Registry` contract.

This contract is owned by the `Timelock` contract, which is the one responsible, through governance actions, to execute proposal that will set contract addresses in the `Registry` contract.

Incentivizer

The `Incentivizer` contract is a general mechanism forked from the Synthetix project which aims to let users stake an `underlyingToken` and receive rewards from it in the form of a `rewardToken`, that can be either different or equal to the first asset.

Both parameters can be updated to new values before the reward program ends by calling the `updateRewardProgram`.

Users can `stake` underlying tokens, `withdraw` parts of them, `claim` all accrued rewards, or call the `exit` function that will withdraw and claim all the assets at the same time.

Eventually, there is also the `rescue` method that will send stucked `ERC20` tokens, or a surplus of reward tokens, to the `Reserve` contract described below.

Migrator

The `Migrator` contract is in charge of handling the conversion of `ESD` tokens from v1 to `ESDS` governance tokens in a fixed ratio,
while bonded `ESD` in the previous version will be converted to `ESDS` into a 1-to-1 ratio.

Users from v1 will have to redeem their coupons before migrating to the new version since the `Migrator` contract will not take them into account. All v1 assets which are migrated are also burned and the contract must be funded with enough `ESDS` beforehand to supply the demand. Several of actions must be performed before migrating the assets, including execution of governance proposals in the governance system of the v1, which were left out of scope.

Vester

The `Vester` contract is a general purpose contract inspired by a deprecated `OpenZeppelin contract's draft` that is able to release to a `beneficiary` an amount of tokens over the course of the time. It is intended to be used with governance `ESDS` tokens and the releasing schedule is linear on time.
The `beneficiary` can transfer its status to another address or withdraw the vested tokens at any time. Finally, the `beneficiary` can also `delegate` the voting powers associated with the vested and unvested tokens to a `delegatee` making the tokens usable while being under the vesting period.

Reserve

- It manages market orders of arbitrary tokens pairs that the governance can list to let users trade against them.
- It can mint and burn both `ESD` and `ESDS` tokens. This contract will be the owner of those tokens contract and it is owned by the governance.
- It handles the state of a borrower party, namely the `Stabilizer` described below, that will borrow `ESD` unbacked tokens, generating debt inside the `Reserve`.
- It handles debt modifications associated to the minting and burning of `ESD` together with the deposits and withdraws of `USDC`.
- It will deposit owned `USDC` on a Compound's vault obtaining `cUSDC` tokens in exchange. It will also redeem and supply more `USDC` to the vault on-demand, whenever needed. It will also claim `COMP` rewards accrued for depositing `USDC` in the vault.
- It is upgradeable, together with the `Stabilizer`.

Users can interact with the `Reserve` through the following methods:

- `mint`: will mint 1 `ESD` for each `USDC` provided in the function call.
- `settle`: will burn `ESD` and transfer back to the user the same amount in `USDC`. This method can only be called if the same amount of burned `ESD` can be deducted from the borrower debt without underflowing.
- `redeem`: will instantly burn an amount of `ESD` tokens and give back to the user an amount of `USDC` that depends on a `redeemPrice` which finally depends on the `reserveRatio` between `ESD` and `USDC` reserves and on a fixed and artificial `redemptionTax`.

Stabilizer

The `Stabilizer` is another core module for the entire protocol. As the `Reserve` does, it is created by composition of several subcontracts. It is in charge of taking out from circulation `ESD` while holders of `sESD` will increase over time their value in `ESD`.

- It uses the result of the `Oracle`, described below, with an `exponential moving average EMA` function to look up at the price of the `ESD/USDC` market pair from a `UniswapV2 Pool`.

- It will borrow unbacked `ESD` tokens from the reserve in an amount calculated by the use of the market price given by the `Oracle` periodically over time.
- It let users redeem accrued rewards in the form of `ESD` tokens by providing their `sESD`. The `redeemAmount` amount will definitively depend on the ratio of the balances of `USDC` and `ESD` of the contract.

Users can interact with the `Stabilizer` through the following methods:

- `supply`: will transfer into the contract an amount of `ESD` tokens and will mint back to the user an amount of `sESD` dependent on the ratio between `sESD` and `ESD` supplies.
- `redeem`: will burn an amount of `sESD` tokens and will transfer back to the user an amount of `ESD` dependent on the ration between `ESD` and `sESD` supplies.
- `redeemUnderlying`: will act the same as the one above but the user will specify the amount directly in `ESD` and the contract will calculate how much `sESD` will have to be burned from the user.

Oracle

The `Oracle` contract is an implementation of a UniswapV2 TWAP oracle. It will be owned by the `Stabilizer` and, once setup, will retrieve the average time-weighted price from the `UniswapV2 ESD/USDC` pool.

Summary

While we were auditing the code, the Empty Set Squad team detected and fixed an issue. The `Reserve` contract was using the `ReentrancyGuard` contract provided by OpenZeppelin, but being this a contract not compatible with the upgradeable pattern chosen by the developer team, the code has been refactored in PR#1.

***Update**: After the audit ended, the Empty Set Squad team found and fixed an issue on pull request 3. In the `ReserveComptroller` contract, if an user tries to mint a small number of `ESD` tokens then the rounding operation could allow the user to mint `ESD` without paying any collateral. However, this operation is capped by `10^12 - 1 wei` of `ESD`, which represents a very small amount.*

parameters, factors that cannot be battle-tested in an early stage of the protocol, the mechanism could increase even further the ratio of supply to collateral. More reasons behind the removal of the module, along with other changes of the codebase, can be found in pull request 19. Due to this, some findings that point to code that has changed may not be valid or accurate. In particular, issues ***M01****,* ***M10****,* ***M11****,* ***L02****,* ***L04****,* ***L14****,* ***N02****,* ***N07****, and* ***N08*** *have been affected. It is worth mentioning that the* `redemptionTax` *variable has been removed from the contracts and it is not used anymore in the calculation of the* `redeemPrice` *value, and that with the removal of the Stabilizer module, the* `ReserveComptroller` *contract reduced its functionalities to only mint new* `ESD` *and to redeem them for a* `redeemPrice <= 1`*. The Stabilizer module was an example of a possible mechanism that could be added into the protocol to achieve the desired peg and stability, and it could be added in a near future once the protocol has been tested in real-time scenarios.*

This audit was not meant to put attention into the game theory design and its incentives to stabilize the final price of `ESD`. We assume proper due diligence in initializing the contracts and setting the right permissions and supplies to the corresponding parts of the protocol.

Overall, we found the code to be clear to follow and read, with most functions and contracts properly documented. However, the lack of an extended supportive documentation made difficult the understanding of some parts of the protocol. We highly recommend the team to improve the supporting documentation to describe every single contract and function. Moreover, there was a lack of comprehensive tests that could tackle edge scenarios and a coverage report. Furthermore, we strongly suggest to run economic simulations that could validate the hypothesis behind the code base, and test a mock-up version of the protocol in a controlled environment such as a public test network.

The code has been audited during the course of 3.5 weeks by two auditors and here we present our findings.

Critical severity

None.

The `Incentivizer` contract has the `rescue` function that can be called to send to the `reserve` any ERC20 token that is left in the contract.

The fact that an arbitrary `token` address is passed proves that it is intended to move all types of ERC20 tokens.

However, the call in the `rescue` function of the `verifyRewardBalance` function will revert anytime there are no sufficient reward assets in the contract, as required in lines 193-194.

The docstrings of the function states that this function can transfer any ERC20 token, but this is not always true if not enough rewards of `rewardTokens` are in the balance of the contract, even though the tokens to be rescued do not count for that balance.

Consider either restricting the `rescue` function to only exclusively rescue `rewardsTokens` or to exclusively call the `verifyRewardBalance` if the tokens to be rescued are reward tokens.

***Update**: Fixed on pull request 14. The EmptySetSquad team stated the following for this issue:*

> *verifyRewardBalance() couldn't ever fail unless either token is rewardToken or the contract has been hacked and drained, this is purely an extra defensive check rather than something that could ever be hit under normal usage.*

[H02] Test are not following contracts logic

The code base presents tests in the protocol/test directory.

Almost all modules (except for the `Governance` contracts along with all libraries) have unit tests under their respective folders and the execution shows that all test pass.

However, some of the tests are not following the logic of the contracts implementation and are not testing appropriate conditions that should revert contract calls.

One example is the `ReserveSwapper` test file.
According to the contract implementation, the `swap` function of the `ReserveSwapper` contract

However, in the corresponding test of this function , the `ESD` address is passed as parameter and the function doesn't revert in this case.

The test passes only because the `ESD` address hasn't been set in the `Registry` contract at test execution time, so the line 75 is comparing the passed input address with the zero address.

In this specific case, the test that actually sets the `ESD` address in the `Registry` contract is the `ReserveComptroller` test suite.

This is considered a major shortcoming in the project, as there is no way to determine if the current implementation matches the system's expected behavior.

Consider being careful when coding tests to validate the expected behavior even on edge cases.

***Update**: Fixed on pull request 16.*

Medium severity

[M01] Governance can be tricked into performing external calls to a malicious contract

The `Incentivizer` contract allows the protocol to give rewards to users that stake underlying assets.

In case of having unused assets by the `Incentivizer` contract, the governance can call the `rescue` function transferring those into the reserve.

However, this opens a backdoor which a malicious user could exploit. An user could create a malicious contract that simulates to be a ERC20 compliant token that deposits tokens in the `Incentivizer` contract, and when the governance passes a proposal to rescue those tokens and send them into the reserve, an external call to the malicious contract will be executed without checks on the other end.

Furthermore, once received into the reserve, the governance could create an order to swap them and when the attacker makes the call to the `swap` function from the `ReserveSwapper`

funds cannot be moved with this attack, reducing the attack surface on the code, and specially on contracts that handle the assets such as the `ReserveSwapper` and the `Incentivizer` contracts, is always recommendable to prevent that future versions of the protocol could be affected by this attack.

Moreover, as discussed in the introduction, the `ReentrancyGuard` shield has been removed from the code while the audit was being performed.

Consider restricting the possibility to perform external calls to untrusted contracts to reduce the attack surfaces in the protocol.

***Update**: Acknowledged. The EmptySetSquad team statement for this issue:*

> *Updated docs with a guide on acceptable ERC20 properties for governance.*

[M02] Documentation issues

Overall, the code base presents well documented functions and clear docstrings. However, there are some designs choices and parts of the code which lack of a proper description.

The `ReserveSwapper` contract allows the governance to list orders on ERC20 token swaps. Any user can trade against those orders once listed. Tokens to be swapped must be owned by the reserve itself but
how this contract is going to be used and which ERC20 are going to be traded is not documented anywhere. Moreover, in line 51 from `ReserveSwapper.sol` it is clear that the reserve can have an unlimited `amount` of tokens to be sold but it's not clear nor documented this design choice and in which cases the governance can list an unlimited amount order for a specific token. Consider adding proper documentation about the intentions over the use of this feature.

The `Stake` contract defines all the properties of the `ESDS` governance token. Balances and allowances for this token are accounted as `uint96` variables in order to pack the `CheckPoint` struct into 128 bits.
Because of this, every place in the code base that makes use of the `ESDS` tokens must carefully manage big balances or allowances in order to avoid overflows. As an example, the `ReserveIssuer` contract is defining the `mintStake` and `burnStake` functions that

developers and users from performing miscalculations.

The `redeem` function from the `ReserveComptroller` contract retrieves `USDC` tokens in exchange of `ESD` tokens, which these are then burned.
Because of the `redemptionTax` that can be set to artificially manipulate the price, the actual redeemed amount, can be artificially lowered to any value, including zero.
Consider explicitly warn users about this possibility and properly document this behavior.

In `line 104-107` from `ReserveComptroller.sol`, the docstrings are not clear about the fact that the user must first approve the contract to transfer tokens.
Consider adding this to the docstrings of the function.

***Update**: Fixed on pull request 15.*

[M03] Missing test coverage report

There is no automated test coverage report. Without this report it is impossible to know whether there are parts of the code never executed by the automated tests; so for every change, a full manual test suite has to be executed to make sure that nothing is broken or misbehaving.

Consider adding the test coverage report and making it reach at least 95% of the source code.

***Update**: Acknowledged. The EmptySetSquad team statement for this issue:*

> *Won't currently fix – Solidity coverage tool currently incompatible with our setup.*

[M04] Orders registration can be frontrun

The governance can call the `ReserveSwapper` contract to register a market order for a token pair. Once the order is registered, users can start calling the `swap` function and exchange one token for the other.

If the governance tries to update an already existing order, an order with a limited amount of tokens, that call can be frontrun by a malicious actor, which can call the `swap` function for the same token pair, zeroing the order amount in the accounting balance.

are not taken into account when registering the order's amount.

Consider checking if the contract has enough balance of the respective tokens before setting any non-limited order amount.

***Update**: Acknowledged. The EmptySetSquad team statement for this issue:*

> *Won't fix – generally speaking, this module is meant to be holdover until we can create a more robust DAO treasury management system as part of a larger effort. Included note in governance on how to properly account for these limitations.*

[M05] Same transaction on different proposal could revert

The `GovernorAlpha` contract implements the functionality to propose, vote, and execute proposals, among other actions.

Each proposal can have multiple transactions to process when it is being executed. When a proposal succeeds, these transactions are queued in the `Timelock` contract by passing all the transaction's information and the time in which the transaction will be able allowed for execution.

However, because different proposals could have one same transaction with the same parameters among many different transactions, if two proposals are queued under the same block, then the first time in which the transaction could be executed will be the same, meaning that the hash used to identify that transaction will be taken. Therefore, during the queue process of the second proposal, the requirement that checks if it was already queued will revert.

Consider adding a proposal nonce as part of the hashed data to identify identical transactions as different ones on different proposals.

***Update**: Acknowledged. The EmptySetSquad team statement for this issue:*

> *Won't fix – safe to assume two proposals will never need to be committed in the same block, and we don't want to modify widely-used forked code unnecessarily.*

[M06] Underlying tokens could get stuck

However, because the contract requests underlying assets to be transferred into the contract in order to proceed, and due to human nature, users may transfer directly those underlying tokens with a simple ERC20 transfer transaction instead of calling the `stake` function.

Although the contract has a function that can be used to move any token into the reserve, the `rescue` function, this function works with any token except the underlying asset. Therefore, any underlying asset sent through a regular transfer transaction will get stuck in the contract without any possibility of taking it back.

Because underlying assets may be sent directly into the contract on a daily basis , consider implementing the possibility to withdraw those stuck assets from the contract.

***Update**: Fixed on pull request 14.*

[M07] Deprecated TokenVesting draft contract is used

The `Vester` contract is using the `TokenVesting` contract provided in a previous deprecated draft version provided by OpenZeppelin.

Using deprecated contracts which are in draft status and not properly audited can be dangerous and increase the attack surface.

Consider properly documenting this and correctly inform users of its use.

***Update**: Acknowledged.*

[M08] Unnecessary ABIEncoderV2

The `ProxyRoot` and `Registry` contracts are using the `experimental ABIEncoderV2` `pragma` directive but there is no explicit use of it in the contract.

Furthermore, the usage of the `experimental ABIEncoderV2` is discouraged in older Solidity versions since there have been important fixes for this encoder since then. The risk can be mitigated by being extra thorough on the testing process of the project at all levels. However, even with great tests there is always a chance to miss important issues that will affect the project.

Update: Fixed on pull request 13. The `ProxyRoot` contract has been removed along with the usage of the `ABIEncoderV2` pragma in the `Registry` contract.

[M09] Untested custom SafeMath library in use

Several contracts implement arithmetical operation functions based on OpenZeppelin's SafeMath library, customized to support safe mathematical operations for `uint256`, such as the `GovernorAlpha` contract, `uint96`, such as the `Stake` contract. Yet, none of the functions are tested, which may hinder users and developers' trust in this custom library. Moreover, any change in the library is not going to be detected by the current test suite, rendering all business logic depending on it vulnerable to potential security issues introduced by these future modifications.

Consider including thorough and extensive unit tests for those functions.

***Update**: Acknowledged. The EmptySetSquad team statement for this issue:*

> *Won't fix – widely-used forked code already tested externally.*

[M10] Updating storage variables could soft-brick the protocol

The `Implementation` contract has the `setRegistry` function which is used to update the address of the `Registry` contract in all contracts that inherit from the `Implementation` contract, such as the `ReserveAccessors` contract contract or the `StabilizerAccessors` contract.

In Line 78 of the same function, the require statements force either:

- To have not set the `registry` address yet.
- That the new `Registry` which is going to be set, has the same `timelock` address of the old `Registry` contract.

The `Registry` contract is owned by the `Timelock` contract, that executes transaction to set addresses on the `Registry` contract on behalf of the governance.

If this happens, the `setRegistry` function of the `Implementation` contract will revert at any call trying to update the `Registry` contract's address, freezing entirely the functionality.

Consider validating that the `timelock` address is a valid contract address whenever it is set in the `Registry` contract. This can be done using the OpenZeppelin `Address.isContract` function or eventually, consider also implementing the `EIP1820` for this use case.

***Update**: Fixed on pull request 12.*

[M11] Possible compromised storage due to hierarchy composition

The design chosen for the upgradeability of certain contracts, such as the `Reserve` and the `Stabilizer` contracts, consists of composing those contracts by means of smaller ones and use an upgradeable proxy pattern with the resulting contract.

For this reason, the storage variables of both modules are stored respectively in the `ReserveState` and the `StabilizerState` contracts.

The intention is to have a unique contract for all the state variables of each module and have less overhead when upgrading through proxy patterns. This can be a good design decision whenever the composition of the contracts is strictly followed in order.

However, there are several cases where some contracts extend and import the same contracts multiple times and in different orders. Some examples are:

- The `StabilizerImpl` contract extends the `Implementation` contract, however it is already been inherited by the `StabilizerAdmin` contract through the `StabilizerComptroller` contract.
- The `StabilizerState.sol` file has duplicated imports. Moreover the `Implementation` contract already imports the `IImplementation` interface.
- The `ReserveImpl` contract imports are already imported in `ReserveIssuer` and `ReserveSwapper` contracts. Moreover, the `Implementation` import is not used.
- All imports of the `ReserveSwapper`, except for the `ReserveComptroller` import, are already imported in the `ReserveComptroller`.

Changing the order of the composition can result in different contracts layouts if imports are mixed in different orders. However this is unlikely to happens since the storage is centralized in one unique contract.

Consider reviewing all orders imports and duplications to ensure a clear composition order and reduce the possibility of compromising storage when upgrading the `Reserve` or the `Stabilizer` implementations.

***Update**: Fixed on pull request 11.*

[M12] Wrong arithmetic operation when normalizing prices

The `Oracle` contract implements the logic to retrieve the updated price of a listed asset.

When a price is updated, the `_normalize` function is called to adjust the difference between the decimals of the pair of assets involved. The value of 6 for `USDC` is compared against the one for the queried token and the price value is multiplied by the difference between each other.

However, when the other token has less than 6 decimals, the price is also multiplied by the difference of decimals instead of dividing by it.

Although currently the other token used will be `ESD` with 18 decimals, which means that it will not encounter this issue, consider dividing instead of multiplying the difference of decimals when the token has less than 6 decimals.

***Update**: Fixed on pull request 10.*

Low severity

[L01] Lack of guardian-role transfer

The `guardian` address from the `GovernorAlpha` contract has a special role inside the system, which is allowed to either cancel a proposal or proceed with the admin-change process, among others.

Consider creating a function to transfer the `guardian` role to another address or consider documenting this limitation.

***Update**: Acknowledged. The EmptySetSquad team statement for this issue:*

> *Won't fix – don't want to modify widely-used forked code unnecessarily.*

[L02] ESD price defaults to one

In the `StabilizerComptroller` contract, whenever the oracle becomes unhealthy, the protocol sets immediately the `ema` to 1.

Whether this choice is intended to protect or stabilize the price of the `ESD` to a fixed price, it is not clear and straightforward why that would be the best mitigation to an unhealthy oracle.

Consider properly justifying this choice or the assumptions that are taken in place whenever the oracle becomes unhealthy.

***Update**: Acknowledged. The EmptySetSquad team statement for this issue:*

> *Expected behavior since a stablecoin's neutral price is $1.00.*

[L03] ERC20 compliant assets may not be used

In the `Oracle` contract, the `_normalize` function is used to normalize the difference in decimals between USDC and a certain token. Although it is meant to be used against ESD, the `Oracle` contract generality allows other assets to be queried there.

Nevertheless, because the ERC20 `decimals` getter is optional, using any asset without the getter in the protocol will fail at retrieving the number of decimals used for their internal accounting, and with it the transaction will revert.

Consider implementing the functionality to allow the usage of ERC20 compliant assets that do not have a `decimals` getter or taking this into account when whitelisting assets.

***Update**: Acknowledged. The EmptySetSquad team statement for this issue:*

The `Implementation` contract declares and implements the logic to get the respective logic's implementation and the admin's address.

However, there are some function declarations that lack its definition, such as setting the registry's address or the owner of the contract.

These two functions are declared in the `StabilizerState` and in the `ReserveState` contracts, and both have the same functionality.

Whether this is meant to be consistent to keep all state variables in the respective contracts or not, having the implementation of a function in different contracts without a proper reason produces a more complex code and lower its readability.

To improve the code's readability and factorization, consider defining both functions in the `Implementation` contract.

***Update**: Fixed in pull request 20. The `IImplementation` interface has been removed, while the `Implementation` contract has been refactored to host definition and implementation of the getters and setters of the `owner`, `registry` and `notEntered` parameters. The `Implementation` contract does that by an unstructured pattern of choosing specific slots to store and set the variables. Since the `ReserveAdmin` and the `StabilizerAdmin` contracts are already extending from the `Implementation`, the `StabilizerAccessors` and `ReserveAccessors` contracts (that extends from those) have been refactored too.*

[L05] Hardcoded parameters in Governance

In the `GovernorAlpha` contract, the `quorumVotes`, `proposalThreshold`, `proposalMaxOperations`, `votingDelay`, and `votingPeriod` functions are returning hardcoded values for the corresponding parameters.

Having hardcoded parameters forces developers to deploy a new contract if one of those parameters needs to be changed.

that can change [illegible] already.

For this reason, consider adding proper setter functions to those parameters. These functions could be called by the `guardian` of the `GovernorAlpha` contract or by the `Timelock` contract through a proposal.

***Update**: Acknowledged. The EmptySetSquad team statement for this issue:*

> *Won't fix – don't want to modify widely-used forked code unnecessarily.*

[L06] High value ERC20 operations may revert

The `ESDS` token is the token used to vote proposals in the governance and it stores its total supply under an `uint256` variable. However, the accounts balances are stored in `uint96` variables.

Furthermore, the `Vester` contract is in charge of gradually distributing ERC20 tokens, such as the `ESDS` token, during the lifespan of the contract. Although the contract has been implemented using `uint256` variables, the same outcome as mentioned would happen when balances come close to the maximum number of an `uint96`.

If external projects adopt the protocol, the expected behavior of a complaint ERC20 may result in unexpected transaction reverts.

Consider either using `uint256` variables for the balances or documenting the reason to use `uint96` variables.

***Update**: Acknowledged. The EmptySetSquad team statement for this issue:*

> *Won't fix – don't want to modify widely-used forked code unnecessarily.*

[L07] Implicit casting

The lack of explicit casting between different variable types, hinders code's readability, making it more error-prone and hard to maintain.

In the same way, in `Stake.sol`, in line 108 and 126 implicit castings of `amount` from `uint96` to `uint256` are performed. Finally in line 179 of the same contract, the returned value should be a `uint256` but it's used an `uint96`.

Consider explicitly casting all integer values to their expected type when sending them as parameters of functions and events. It is advisable to review the entire codebase and apply this recommendation to all segments of code where the issue is found.

***Update**: Acknowledged. The EmptySetSquad team statement for this issue:*

> *Won't fix – don't want to modify widely-used forked code unnecessarily.*

[L08] Lack of event emission after sensitive actions

The `__acceptAdmin` and the `__abdicate` functions of the `GovernorAlpha` contract do not emit relevant events after executing their sensitive actions.

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

***Update**: Acknowledged. The EmptySetSquad team statement for this issue:*

> *Won't fix – don't want to modify widely-used forked code unnecessarily.*

[L09] Lack of input validation

There are several places in the code base where input parameters are passed in function calls without any kind of validation of their values. Examples are:

- The `amount` variable of the `stake` and `withdraw` functions from the `Incentivizer` contract is not validated to be non zero.
- The `propose` function from the `GovernorAlpha` contract is not checking whether the `description` is empty or not.
- The `approve` function from the `Stake` contract is not validating the `spender` input parameter.

Consider adding proper checks on the values passed in function calls.

***Update**: Partially fixed on pull request 9. The `GovernorAlpha` and `Stake` contracts have not been changed. The EmptySetSquad team statement for this issue:*

> *Leaves GovernorAlpha and Stake as-is to not modify forked code unnecessarily.*

[L10] Missing tests

No tests were found for the governance system implemented in the contracts. This is considered a major shortcoming in the project, as there is no way to determine if the current implementation matches the system's expected behavior.

Consider adding tests for diverse cases and edge scenarios.

***Update**: Acknowledged. The EmptySetSquad team statement for this issue:*

> *Won't fix – widely-used forked code already tested externally.*

[L11] View function creates a pointer to storage

The `GovernorAlpha` contract implements the functionality to propose, vote, and execute proposals.

To know what a proposal will execute, an user can call the `getActions` function which will retrieve all the information of the transactions.

Although this function is `public view` and it is not meant to change the data in storage, the pointer created points to storage instead of memory.

Whether this is a design choise or not, consider changing it to memory to reduce the attack surface in the contract.

***Update**: Acknowledged. The EmptySetSquad team statement for this issue:*

> *Won't fix – don't want to modify widely-used forked code unnecessarily.*

current accumulated price and the stored one by the elapsed time, and then dividing again by `2**112`, using the `ratio` method from the `Decimal` library, to get the output in price units.

However, the precision of the outcome could be increased by altering the order of the operations by multiplying the time elapsed and the `2**112` factor to then take the ratio of it, which internally multiplies first the numerator with the `BASE` constant.

Consider changing the order of the arithmetic operations to improve the outcome's precision.

***Update**: Acknowledged. The EmptySetSquad team statement for this issue:*

> *Won't fix – effect on oracle is negligible, so we'd prefer to leave as is to increase readability by separating the numeric computation from the base conversion.*

[L13] Centralized registry is not adopted in the whole code base

Having the `Registry` contract allows the protocol to have a unique source of information of the key components of the system.

Coupled together with the fact that the `Registry` contract is triggered by the governance, this should ensure a robust system to have system contracts depending on a registry where addresses can be updated anytime through a proposal.

However, not all contracts are using the `Registry` contract and its specific state variables. Some examples are:

- The `stake` and `timelock` variables from the `GovernorAlpha` contract.
- The `reserve` variable from the `Incentivizer` contract.

Consider using the `Registry` contract in all the contracts that need to read system contract addresses to improve consistency and to have a homogeneous functionality in the system.

***Update**: Partially fixed on pull request 8. The `GovernorAlpha` contract does not inherit the functionalities of a centralized registry from the `RegistryAccessor` contract.*

[L14] Re-implementing ECDSA signature recovery

OpenZeppelin contracts, which has been audited and is constantly reviewed by the community.

Consider importing and using the `recover` function from OpenZeppelin's ECDSA library not only to benefit from bug fixes to be applied in future releases, but also to reduce the code's attack surface.

***Update**: Acknowledged. The EmptySetSquad team statement for this issue:*

> *Won't fix – don't want to modify widely-used forked code unnecessarily.*

[L15] New Oracle instance has to be deployed after a new StabilizerImpl deployment

The `Oracle` contract has the functionality to adapt and retrieve the prices for the `ESD` token. It is owned by the `StabilizerComptroller` contract.

To set the configuration on both contracts, the owner of the `StabilizerComptroller` contract has to call the `setup` function which would set as one the conversion between `ESD` and `USDC` but also it will call the `setup` function from the `Oracle` contract. There, the new pair will be created and it will be stored in the variables.

However, in case a new `StabilizerComptroller` is deployed, the setup of it will not be able to succeed by using the same `Oracle` contract, because a requirement that checks if the pair has been already setup for `ESD` will revert, preventing to set as one the moving average for that token.

Consider either allowing to set the initial value of the moving average in the `StabilizerComptroller` when deploying a new version of it without deploying a new `Oracle` contract, or documenting the reason to setup both contracts under the same call.

***Update**: Acknowledged. The EmptySetSquad team statement for this issue:*

> *Won't fix – this behavior is ok.*

[L16] Uncommon ERC20 are not managed

The `ReserveSwapper` contract allows the `Governance` to list limit orders of ERC20 tokens.

However, this internal fees are not being handled by the [illegible] contract.

Therefore, the governance should be aware that the protocol currently does not fully support tokens that charge fees during transfers, and whitelisting such tokens can result in unexpected behavior, such as insolvency. If the protocol is expected to be compatible with these type of tokens, consider modifying the internal accounting mechanism to use the actual amount of assets deposited. One plausible approach would be to query the contract's token balance before and after the execution of the token transfers to obtain the real amount of tokens deposited. For reference, consider how Compound's protocol handles this scenario in their CErc20 contract.

***Update**: Acknowledged. The EmptySetSquad team statement for this issue:*

> *Updated docs with a guide on acceptable ERC20 properties for governance.*

[L17] Unhandled return value

The `execute` function from the `GovernorAlpha` contract is internally calling the `executeTransaction` function from the `Timelock` contract.

The `executeTransaction` function returns the data returned by the contract call in the `returnData` parameter, but this parameter is not read nor stored anywhere in the `GovernorAlpha` contract.

Whether the `returnData` is needed or not, consider reading or storing it. Alternatively, consider modifying the `executeTransaction` function to not return any parameter at all if it is not needed or documenting the reason to not use its returned value.

Similarly, the `delegate` function from the `Stake` contract is declared as not returning any value, but it is actually using the `return` keyword in its implementation.

Consider removing the `return` instruction.

***Update**: Acknowledged. The EmptySetSquad team statement for this issue:*

> *Won't fix – don't want to modify widely-used forked code unnecessarily.*

The `StabilierToken` contract implements the functionalities of the `sESD` ERC20 token.

One of those is the `transferFrom` function that allows to transfer tokens on behalf of another user when the allowance is properly set.

However, if a non-allowed user tries to transfer zero number of tokens on behalf of another another account by calling the `transferFrom` function, the transaction not only will succeed but it will trigger both `Transfer` and `Approval` event emission, meaning that the malicious user could disguise this harmless transaction as an approval on a non-owned account.

Consider requiring a non-zero value transfer when using the `transferFrom` function to prevent users from being deceived and to improve the readability of off-chain services that track event logs.

***Update**: Acknowledged. The EmptySetSquad team statement for this issue:*

> *Won't fix – will stick with current behavior to conform to OZs implementation of ERC20.*

Notes & Additional Information

[N01] Not following the Checks-Effects-Interactions pattern

The `Dollar` contract extends the functionality of the `ERC20Detailed` and `Permittable`.

The `transferFrom` function check if the sender is allowed by the holder to send the tokens but this check is done after the transfer has been made.

Solidity recommends the usage of the Checks-Effects-Interactions Pattern to avoid potential security issues, such as reentrancy.

Although in this scenario it does not represent a possible security issue, consider always following the "Check-Effects-Interactions" pattern, thus checking the parameters before modifying the contract's state.

***Update**: Acknowledged. The EmptySetSquad team statement for this issue:*

There are some examples in the code base where gas consumption can be optimized by returning or failing earlier.

- The requirement performed on Line 287 of `ReserveComptroller.sol` can be done after line 281 using the `newBorrowed` variable.
- The order of lines 94-95 of `StabilizerToken.sol` can be inverted to fail earlier.

Consider reviewing the code base for occurrences like these and refactor the code to be more gas efficient.

***Update**: Partially fixed on pull request 7. Only the operation order in the `StabilizerToken` contract was changed.*

[N03] Modifier could replace repeated requirements

In the `GovernorAlpha` contract there are certain functions that can only be called by the `guardian` address. These are to cancel a proposal, accept an admin, abdicate the guardian role, set the pending admin, and execute the pending admin process.

On all of these functions, the requirement is copied instead of using a modifier that restricts the call of the functions to only the guardian. This not only cost more at the time of the deployment but also renders the readability of the code.

Consider replacing all the guardian only-access requirements to a modifier that accomplishes the same functionality.

***Update**: Acknowledged. The EmptySetSquad team statement for this issue:*

> *Won't fix – don't want to modify widely-used forked code unnecessarily.*

[N04] Implicit and default values

In the code base, there are several occurrences of parameter's values that imply a certain purpose.

Similarly, the `Pending` status of the `ProposalState` struct from the `GovernorAlpha` contract is the default value of the enum and it is being used to tag a pending proposal status instead of having a non-existent proposal state.

To improve the code's readability, consider replacing hardcoded values for explicit constants that properly describe their purposes. Consider also not using the default values for specific meanings, as default values are the ones that appear before an object is initialized, which may lead to confusion.

***Update**: Acknowledged. The EmptySetSquad team statement for this issue:*

> *Won't fix – don't want to modify widely-used forked code unnecessarily.*

[N05] Inconsistent coding style

In the code base, there are some inconsistencies in the style used for the code. Some examples are:

In the `Incentivizer` contract:

- The `verifyRewardBalance` function is private and placed in the // ADMIN section. It should have a starting `_` in the name as it is done for other private functions and being placed under a new "// PRIVATE" section.
- The `settle` and `settleAccount` functions are internal and placed under the // FLYWHEEL section, but then, internal functions are defined under the // INTERNAL section and all start with a `_`.

In the `ReserveComptroller` contract:

- In line 183 it is used a variable to store the `ESD` address but it is not done the same on line 195 where the output from the registry is used directly. Furthermore, the `IERC20` and `IManagedToken` interfaces are used for the same contract. In this case, consider merging and standardizing the two interfaces if they are used exclusively by the same contract.

proposal in one line, however the `cancel` function does the same but in 2 lines.
- Revert messages of the `GovernorAlpha` follow a design which is not used by all other contracts.

In order to improve readability, consider following a unique consistent style while coding the contracts.

***Update**: Partially fixed on pull request 18. The EmptySetSquad team statement for this issue:*

> *ReserveComptroller Leaves as-is, the pattern being: don't utilizes intermediary variables unless the variable would be accessed more than once in the resulting code.*
> *GovernorAlpha Leaves as-is to not modify forked code unnecessarily.*

[N06] Lack of indexed parameters in events

There are places in the code base where events defined in the contracts do not index any parameter. Some examples are:

- The `GovernorAlpha` contract.
- The events from the `IImplementation` contract.

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

***Update**: Partially fixed on pull request 6. Events on the `GovernorAlpha` contract are still not indexed.*

[N07] Misleading or erroneous docstrings

In the code base, there are docstrings that may confuse the user or that are incorrect. Some examples are:

- The `IComptroller` interface from the `ReserveVault` contract has a docstring that states `ICErc20` where it should be `IComptroller`.
- Line 133 from `ReserveState.sol` should say "Redemption tax".
- Line 84 from `ReserveComptroller.sol` states "Rhe" where it should be "The".

`setup` function but those are copied from the docstrings of the `rate` function below.

In order to improve readability, consider fixing all incorrect docstrings in the code base.

***Update**: Fixed on pull requests 5 and 1.*

[N08] Several contracts developed per file

In the code base, the `StabilizerState.sol` file together with the `ReserverVault.sol`, `ReserverState.sol`, `Migrator.sol`, `GovernerAlpha.sol`, and `Interfaces.sol` files all present multiple contracts or interfaces definitions inside of the same file.

To improve understandability and readability, but also to have a more modular code base, consider having one contract or interface per Solidity file.

***Update**: Acknowledged. The EmptySetSquad team statement for this issue:*

> *Won't fix directly – significantly cleaned up via pull request 19 and pull request 20.*

[N09] Naming issues

In the code base, there are several occurrences of unclear or confusing names. For example:

- The `GovernerAlpha.sol` file defines a contract called `GovernorAlpha`.
- The `getPartial` function's name of the `Decimal` library doesn't reflect correctly the functionality of the function.
- The name of the `_paid` mapping of the `Incentivizer` contract doesn't suit with the purpose, it is used to track the latest reward per unit instead of the amount of paid rewards as suggested by its docstring.

To improve consistency and readability of the code base, consider changing variable, function, and file names to better reflect their intentions.

***Update**: Partially fixed on pull request 4. The `getPartial` function still has a name that does not reflect its functionality.*

examples are:

- In the GovernorAlpha contract.
- In the ICErc20 interface.

To favor explicitness, consider changing all instances of `uint` to `uint256`.

***Update**: Acknowledged. The EmptySetSquad team statement for this issue:*

> *Won't fix – don't want to modify widely-used forked code unnecessarily.*

Conclusions

No critical and 2 high vulnerabilities were found among other issues with lower severities. Recommendations and comments have been provided to improve code quality and readability and some security measures have been recommended to improve the overall health of the system.

Related Posts

Beefy Zap Audit

BeefyZapRouter serves as a versatile intermediary designed to execute users' orders through routes...

OpenBrush Contracts Library Security Review

OpenBrush is an open-source smart contract library written in the Rust

Linea Bridge Audit

Linea is a ZK-rollup deployed on top of Ethereum. It is designed to be EVM-compatible and aims to...

OpenZeppelin

Defender Platform

Secure Code & Audit
Secure Deploy
Threat Monitoring
Incident Response
Operation and Automation

Services

Smart Contract Security Audit
Incident Response
Zero Knowledge Proof Practice

Learn

Docs
Ethernaut CTF
Blog

Company

About us
Jobs
Blog

Contracts Library

Docs

Privacy | Terms of Use