Audit Report

Zivoe - Core Contracts

Delivered: 2023-07-10

Prepared for Zivoe by Runtime Verification, Inc.

Summary

Runtime Verification, Inc. has audited the smart contract source code of the Zivoe project. The review was conducted from 05-06-2023 to 10-07-2023.

Zivoe engaged Runtime Verification in checking the security of their Zivoe project, which is a international decentralized credit protocol that facilitates on-chain stablecoin lending to regulated entities. Yield is generated from RWA's and brought back on-chain to the protocol, where then the utility is provided to the protocol.

The issues which have been identified can be found in the sections titled Findings and Informative Findings. Issues addressed by the client are identified accordingly with the relevant fixed commit provided.

Scope

The audited smart contracts are:

- `ZivoeDAO.sol`
- `ZivoeGlobals.sol`
- `ZivoeGovernorV2.sol`
- `ZivoeITO.sol`
- `ZivoeLocker.sol`
- `ZivoeMath.sol`
- `ZivoeRewards.sol`
- `ZivoeRewardsVesting.sol`
- `ZivoeToken.sol`
- `ZivoeTranches.sol`
- `ZivoeTrancheToken.sol`
- `ZivoeYDL.sol`
- `libraries`
    - `FloorMath.sol`
    - `OwnableLocked.sol`
    - `ZivoeGTC.sol`
    - `ZivoeTLC.sol`

The audit has focused on the above smart contracts, and has assumed correctness of the libraries and external contracts they make use of. The libraries are widely used and assumed secure and functionally correct.

The review encompassed the `Zivoe/zivoe-core-foundry` private code repository. The code was frozen for review at commit `17c05ef4e5e503e8911674ca7c86cbdc0ddae59c`.

The review is limited in scope to consider only contract code. Off-chain and client-side portions of the codebase are *not* in the scope of this engagement.

Assumptions

The audit assumes that all addresses assigned a role must be trusted for as long as they hold that role. Apart from the deployer that is responsible to create and deploy the contracts, an `owner` role (see [OpenZeppelin's access control]) is present for some of the contracts in the protocol such as `ZivoeGlobals` and `ZivoeDAO`.

The admin addresses of the respective contracts need to be absolutely trusted. A multisig controlled by the Zivoe team will deploy the protocol and start the ITO period. After the ITO period ends, the governance contract operating on a DAO-based model will take over. Furthermore, we assume that the deployers and the governance address take relevant steps to ensure that the state of the deployed contracts remains correct. In addition to setting the correct state, it is also contingent upon governance to maintain a reasonable state. This includes only accepting trustworthy tokens and setting protocol parameters honestly.

Note that the assumptions roughly assume "honesty and competence". However, we will rely less on competence, and point out wherever possible how the contracts could better ensure that unintended mistakes cannot happen.

Methodology

Although the manual code review cannot guarantee to find all possible security vulnerabilities as mentioned in Disclaimer, we have used the following approaches to make our audit as thorough as possible. First, we rigorously reasoned about the business logic of the contract, validating security-critical properties to ensure the absence of loopholes in the business logic and/or inconsistency between the logic and the implementation. Second, we carefully checked if the code is vulnerable to known security issues and attack vectors. Finally, we met with the Zivoe team to provide feedback and suggested development practices and design improvements.

This report describes the **intended** behavior of the contracts under review, and then outlines issues we have found, both in the intended behavior and in the ways the code differs from it. We also point out lesser concerns, deviations from best practice and any other weaknesses we encounter. Finally, we also give an overview of the important security properties we proved during the course of the review.

Disclaimer

This report does not constitute legal or investment advice. The preparers of this report present it as an informational exercise documenting the due diligence involved in the secure development of the target contract only, and make no material claims or guarantees concerning the contract's operation post-deployment. The preparers of this report assume no liability for any and all potential consequences of the deployment or use of this contract.

Smart contracts are still a nascent software arena, and their deployment and public offering carries substantial risk. This report makes no claims that its analysis is fully comprehensive, and recommends always seeking multiple opinions and audits.

This report is also not comprehensive in scope, excluding a number of components critical to the correct operation of this system.

The possibility of human error in the manual review process is very real, and we recommend seeking multiple independent opinions on any claims which impact a large quantity of funds.

Zivoe: Contract Description and Properties

Overview

Zivoe aims to provide a credit access platform that can be backed up by different means of on-chain and off-chain liquidity provision. Zivoe protocol enables on-chain lending and borrowing via different loan models. Zivoe offers three different tokens to facilitate value distribution to the participants of the protocol. Yields obtained by the deposited capital to the protocol are distributed to the capital providers via senior and junior tranche tokens. These tokens represent two different levels of risk associated with the provided credits that are called junior and senior tranches. Tranches segment the credit risk where credit provided by the junior tranche is considered to be of higher risk than the senior tranche. In return for the higher risk taken, junior tranche depositors receive more rewards.

Zivoe also provides governance over DAO model where governance participants are provided with rewards using Zivoe Governance Tokens (ZVE). ZVE holders obtain voting power proportional to the amount of tokens they hold and direct the investment of capital. ZVE stakers also receive rewards proportional to the amount of yield obtained by the protocol either through direct reward distribution or through a vesting procedure.

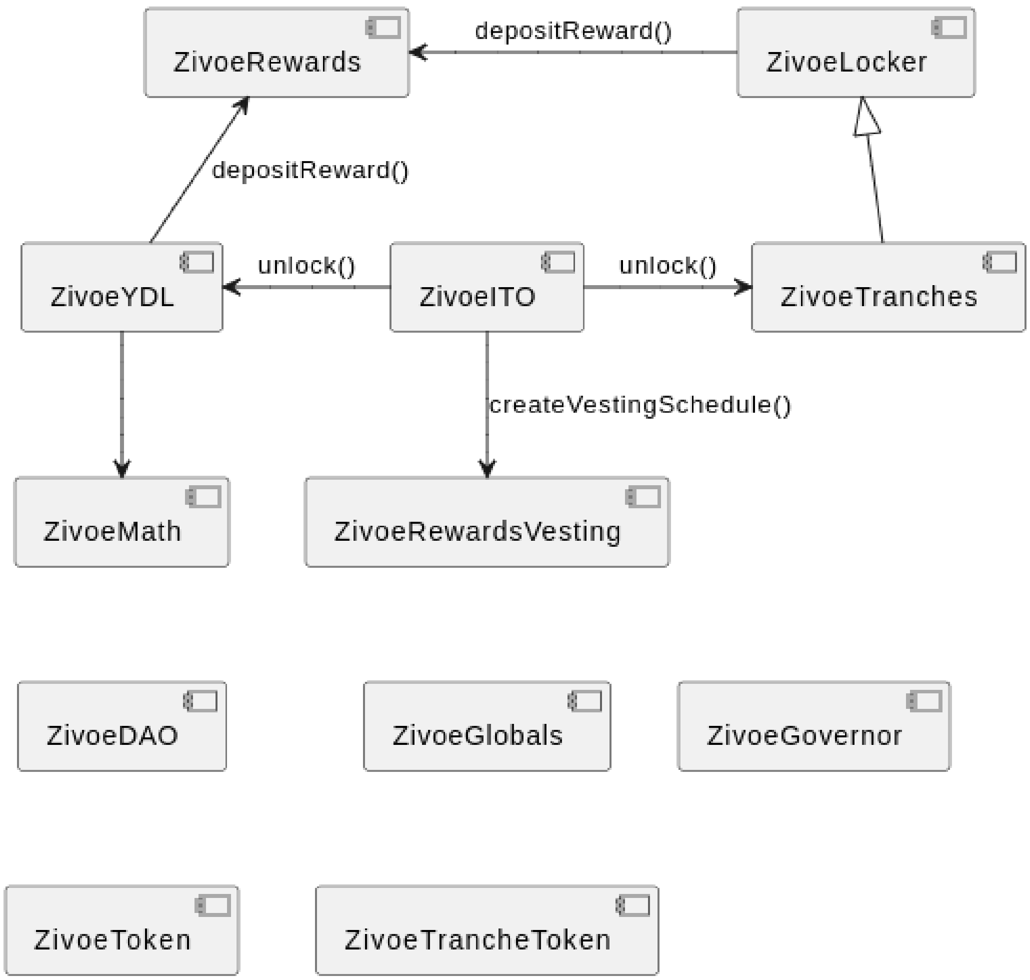
Initially, Zivoe protocol is planned to launch after an ITO period where participants are expected to provide stable coins to the tranches and start collecting rewards after the ITO period ends and the DAO governance is enabled. During the protocol operation, capital organization and yield distribution is organized over a contract family called lockers. Each locker represents a different means of valuation of the capital such as yielding, credit provisioning, treasury organization, etc. Very briefly, DAO interacts with the lockers to govern the capital managed by the protocol during the operation of the protocol. From the codebase perspective, locker contracts are also organized separately from the core modules as mentioned earlier.

Before diving into more detailed description of the protocol, we overview the actors of the protocol as follows:

- Admin (Zivoe Labs): In protocol's current condition ZivoeLabs have the administrative privileges to set the important parameters of the protocol in the ZivoeGlobal contract. Also creating and revoking vesting schedules can be performed by the ZivoeLabs if not by the ITO. During deployment temporary administrative privileges are also attained by the deployer as described below.
- Deployer: During the deployment of the protocol some of the access rights are temporarily attained to the accounts that perform deployment. These include TimeLock admin role for the TimeLock controller contract, ownership of ZVT and DAO contracts, ownership of junior and senior tranche tokens and the ownership of ZivoeGlobals

contract. During deployment, deploying contracts grant these roles temporarily and are expected to perform necessary ownership renouncing and ownership transfer operations before the operation of the protocol.

- Governance (DAO): Governance is carried out via OpenZeppelin's on-chain governance contracts. Proposer role is granted to the ZivoeGovernor contract at deployment time and remaining roles such as executor and Timelock Administrator are organized following on-chain governance process afterwards.
- User: Users provide liquidity to the protocol via different ways such as depositing to tranches and/or staking.
- Borrower: Borrowers take advantage of the loans offered by the protocol. Borrowers should go through a KYC process to be utilized in case of a defaulted loan.
- ITO: There exists a period before the operational activation of the protocol where early participants can invest in tranches and earn rewards. This period is handled by the ITO contract. Even though it is not a temporary role, ITO contract is responsible for unlocking the tranches and yield distributions after the ITO period as well as creating and revoking vesting schedules if not by the ZivoeLabs.
- Locker: Lockers keep assets of the protocol for various different purposes such as accounting loans, handling yield bearing investments, handling redemptions, etc. There exist various different lockers but all of them provide a basic common functionality presented in the ZivoeLockers abstract contract.



We are going to cover the protocol operation by overviewing the responsibilities and operational properties of the contracts. We would like to note that the Zivoe Lockers abstract contract is left to be covered together with the individual lockers.

Before definitions of the core modules we would like to describe how modules interact with each other in a broader context over a typical deployment process. Following Figure presents main core modules and important interactions between them. To simplify this diagram, interactions of the widely used components such as ZivoeGlobals, ZivoeTokens and ZivoeTrancheTokens are omitted. From a very broad point of view it can be seen that after the ITO period ends tranches and yield distribution is unlocked which in turn deposits rewards. Also rewards during the ITO can be directly start vesting by creating a vesting schedule.

During the deployment of the protocol following process is followed:

1. Deploy Zivoe Globals
2. Deploy Zivoe Token (Deployer owns all the initial balance)
3. Deploy governance contract
    a. Deploy ZivoeTLC with proposers and executors

    b. Deploy ZivoeGovernor
    c. Grant proposer role to governance and revoke other roles
4. Deploy ZivoeDAO
    a. Deployer transfers 35% of ZVE to DAO
    b. DAO owner transfers ownership to governance
5. Deploy zSTT and zJTT
6. Deploy ZivoeITO
    a. Set stable coins
    b. Construct ITO contract
    c. Grant ITO contract minting access
7. Deploy Zivoe Tranches
    a. Construct contract
    b. Transfer ownership and 5% of ZVE
8. Deploy Staking contracts through ZivoeRewards
9. Deploy Yield Distribution contracts
10. Deploy Rewards vesting contract
    a. Deployer transfers 60% of ZVE
11. Update globals
    a. Wallet addresses
    b. Stable coins

After briefly overviewing the main core modules, interactions between them and deployment of the modules, we can summarize the operation of each module as follows.

ZivoeGlobals

This contract holds important addresses for the protocol during its operation as well as a couple of functions that are used through the protocol operations. It contains an array of 14 addresses that keep the important contracts. Since this list and the abbreviated variable names are frequently used in the protocol and in this report a table containing those is presented below:

Abbreviation/State variable	Meaning
DAO	Decentralized Autonomous Organization Contract
ITO	Initial Tranche Offerings Contract
stJTT	Staking Contract for Junior Tranche Tokens
stSTT	Staking Contract for Senior Tranche Tokens

stZVE	Rewards Contract
vestZVE	Rewards Vesting Contract
YDL	Yield Distribution Contract
zJTT	Junior Tranche Token Contract
zSTT	Senior Tranche Token Contract
ZVE	Zivoe Token Contract
ZVL	Zivoe Labs Contract
GOV	Governor Contract
TLC	Timelock Controller Contract
ZVT	Zivoe Tranches Contract

Additionally, this contract also keeps a list whitelisted keeper, locker and stable coin addresses to set necessary allowances.

This contract also provides functionality for the following operations:

- Increase and decrease the amount of defaults used in the accounting operations. Defaults can be very basically refined as the loans that have not been repaid according to the lending plan. Throughout the lifetime of a loan it can be temporarily/permanently and partially/completely fall into the default category. This affects the accounting performed by the protocol such as adjusting the supplies in the tranches by taking into account the total amount of defaults.
- Standardize a given amount by considering its decimal precision in the fixed point arithmetic.
- Adjust the amount of supplies according to the total amount of defaults in the protocol. Adjusting initially is performed over the junior supplies by subtracting the amount oıf defaults from the total JTT supply. If the amount of total defaults are larger than JTT supplies, then the rest of the defaults are decremented from STT supplies.

ZivoeITO

This contract governs the “Initial Tranche Offerings” phase of the protocol which is the initial phase to provide assets to the protocol to receive JTT or STT. After the ITO ends, collected assets are transferred to the governance of DAO and ZVE vestings begin to provide governance ability to participants of ITO. This contract basically accepts deposits to either of the tranches until the ITO period ends. Afterwards, it “unlocks” the YDL and ZVT contracts and deems those operable. Also it starts letting the participants begin vesting ZVE rewards.

ZivoeDAO - ZivoeToken

ZivoeDAO is a timelock controlled contract to facilitate the governance of the protocol over decentralized voting mechanisms. Each governor is provided with voting ability proportional to the amount of ZVE tokens they own.

ZVE is an ERC20 (and ERC20Votes) compatible token that is initially minted with 25 million ether of supply and distributed according to the staked amount by the participant of the protocol.

The DAO acts like a proxy for the lockers of the protocol. It performs necessary pull/push functionality to the lockers according to the asset types that the lockers support. Currently pushing/pulling, partial pulling, multi pulling/pushing and partial multi pulling is supported for ERC20 based locker where pushing/pulling, multi pushing/pulling is supported for ERC721 based lockers and only pushing/pulling is supported for ERC1155 based lockers.

ZivoeRewards - ZivoeRewardsVesting

These contracts facilitate staking and reward distribution operations. Both contracts have a number of common functionality they provide. It is possible to add a maximum number of 10 different reward tokens to the contract. These tokens are used in rewards distribution after depositing some rewards to the contract over the reward tokens in this list. Rewards are distributed gradually over time and the properties of this distribution period are set at the time when the reward token is being added to the list of reward tokens. Additionally some properties (such as the rate of rewards being distributed per second) are set when the first reward deposit is made to the contract.

To be able to receive rewards, participants must stake some amount to the staking token designated at contract construction. Additionally, it is possible to stake on behalf of another participant. These two functionality are the main differences of ZivoeRewards compared to ZivoeRewardsVesting where staking is automatically performed.

ZivoeRewardsVesting, as a difference from ZivoeRewards, lets a vesting schedule be applied per participant. During the creation of the vesting schedule the amount staking is accepted and staking is directly performed for a specific account. ZivoeRewardsVesting does not support staking other than a vesting schedule. It is also possible to revoke a vesting schedule. Another difference is it is not possible to use ZVE in this contract as a rewards token.

Both contracts support withdrawal of staked amount, accounting for the necessary updates in rewards distribution.

ZivoeTranches - ZivoeTrancheToken

Governance of tranche operations are performed using these contracts. Tranches themselves are also lockers. During their operation, deposits to junior or senior tranches are accepted, distributing the required amount of reward to the depositor for each deposit. During the deposits contract locks the junior tranche if the supplies in junior tranche surpass a predetermined ratio between adjusted JTT and adjusted STT supplies (adjusted supply is the total supply minus defaults in the system as explained earlier). Tranche tokens are basic ERC20 tokens where the minting capability is limited to a predetermined set of participants given a minter role.

ZivoeYDL - ZivoeMath

As the protocol keeps its operations, assets deposited are used in many different yield distribution opportunities. Obtained yields are distributed to the stakeholder via the ZivoeYDL contract. During the distribution the proportion of the yield to be distributed to the participants of junior and senior tranche is calculated mainly according to the target yield ratio set by the governance. Distribution is performed gradually over a 30 day time period and distribution amount is updated every 30 days. Yields are distributed as rewards to the protocol, senior and junior tranches and other recipients that may be set by the governance.
During yield distribution calculation the distribution is performed according to the amount of successful target yield accomplishment. If the yield surpasses the target the surpassing amount is also distributed as residual yield. During the calculation of these, a number of foımulas, present in ZivoeMath, are used to facilitate the accounting which can be found in the protocol documentation.

Properties

Several important properties should always be satisfied by the contract. Here will only be listed properties related to the contracts under the audit. Properties associated with the use of external libraries are assumed to hold.

In the following, we list several properties that the contract should satisfy. These are not the only ones, but they are fundamental for the correctness of the protocol and, as such, deserve special attention.

ZivoeDAO.sol

1. `TimeLockController` should own it. The ownership must remain the same.
    - `ZivoeDAO` is `OwnableLocked`. In the deployment scheme - `Utility.sol` - the deployer (`owner` of the contract) calls, right after deployment, the function `transferOwnershipAndLock`. This function assigns the `_locked` variable to `true`, which makes it impossible to `transferOwnership` again.

- Note: If using the latest version of OpenZeppelin libraries, the constructor of `Ownable` can receive as a parameter the `TimeLockController`, assigning the `owner` to it. It would still be needed to assign the `_locked` variable to `true` in order to make it impossible to `transferOwnership` again.

2. Only Governance, though `TimeLockController`, should be able to `push` or `pull` assets from a locker.

   - All the functions to `push` and `pull` from the lockers have the `onlyOwner` modifier, which guarantees that only the `owner` of the contract - `TimeLockController` - can call these functions.

3. The lockers to `push` assets must be whitelisted.

   - All the `push` functions ensure that the `locker` to push the assets belongs to the whitelist of lockers through `require( IZivoeGlobals_DAO(GBL).isLocker(locker))`.

4. The lockers to `push` or `pull` assets must be valid to the respective asset being pushed or pulled.

   - All the functions require the corresponding `canPush`/`canPull` functions to return `true`.

5. A `locker` must only be able to `transfer` the exact `amount/token` approved by the governance.

   - The functions `pushERC721` and `pushMultiERC721` set approval for all `tokenIds` before calling the `push` function in the `locker`. So this property may be violated if a malicious `locker` gets whitelisted and upgrades the `push` function to transfer all the tokens.
   - All the functions are protected with the `nonReentrant` modifier, which prevents re-entrancy attacks that could potentially lead to undesired transfers from DAO's capital.

6. The allowance to any `locker` to any asset must be equal to `0` after any contract interaction.

   - The approval to any `locker` is only given in the `push` functions right before the call to `pushToLocker`. Right after the call to `pushToLocker`, the approval is set to `0`. So, after each contract interaction, the approval of ZivoeDAO to any `locker` is `0`.

ZivoeGlobals.sol

1. All the contract variables storing relevant addresses for the protocol contain valid addresses.

   - All the variables are initialized in the function `initializeGlobals`, where there are no checks for the variables being initialized. The deployer's responsible for guaranteeing that the arguments for the `initializeGlobals` function are correct. According to the deployment scheme in `deployCore` in `Utility.sol`, the variables are correctly initialized. However, notice that an unintended mistake can occur in the actual deployment. Except for the `ZVL`, none of the addresses can be modified after `initializeGlobals`.

- The function `transferZVL`, which transfers `ZVL` access control to another `account`, does not check for the `address(0)`. Besides, the new `account` may be an invalid address, in the sense that Zivoe Labs does not have access to it - this can happen if, for instance, the new `address` is mistyped. In both cases, Zivoe Labs loses complete access to all the functionalities restricted to `onlyZVL`.

2. Only whitelisted lockers should be able to increase or decrease the `defaults`.

    - True. Both functions, `increaseDefaults` and `decreaseDefaults`, enforce that only whitelisted lockers can call them through:

```
require(isLocker[_msgSender()],      "ZivoeGlobals::increaseDefaults()
!isLocker[_msgSender()]");
```

3. The function `initializeGlobals` must only be callable once by the contract `owner`.

    - The `ZivoeGlobals` contract is `Ownable`, which means that by default, the contract's `owner` is the deployer. The function `initializeGlobals` ensures that only the `owner` can call it through the modifier `onlyOwner`.

    - However, it is possible to call the function more than once if, in the first call to `initializeGlobals,` the `DAO` parameter, i.e., the argument `globals[0],` is `address(0)`. Again, the deployer's responsible for guaranteeing that this function is only callable once by passing the correct arguments.

4. All contract addresses, except for `ZVL`, should not be possible to modify after initialization.

    - True if property 3 holds because, except for `ZVL`, there are no setters for the contract addresses.

5. `ZVL`, and only `ZVL`, should be able to transfer the `ZVL` access control to a new `address`.

    - Only `ZVL` can transfer access control to a new `address` because the function `transferZVL` ensures that through the modifier:

```
modifier onlyZVL() {
   require(_msgSender() == ZVL, "ZivoeGlobals::onlyZVL() _msgSender()
!= ZVL");
   _;}
```

    - `ZVL` may not be able to transfer access control to a new `address` if 1. is not fixed, i.e., if at some point `ZVL` becomes an invalid `address`.

6. `ZVL`, and only `ZVL`, should be able to update the `keepers` whitelist, the `lockers` whitelist and the `stablecoins` whitelist.

- Only ZVL can update keepers, lockers and stablecoins whitelists because the functions responsible for updating those variables - updateIsKeeper, updateIsLocker and updateStablecoinWhitelist - ensure that through the modifier onlyZVL.
- ZVL may be unable to update keepers, lockers and stablecoins whitelists if property 1. is not fixed, i.e., if ZVL becomes an invalid address.

7. After initialization, the contract should not have any owner.
   - According to the deployment scheme - function deployCore in Utility.sol - the owner renounces ownership after calling initializeGlobals.

ZivoeGovernorV2.sol

1. The _timelock address should always be valid.
   - There are no checks for the address(0) in the constructor of ZivoeGTC. Therefore, the deployer must guarantee that the parameter _timelock to the ZivoeGovernorV2 is correct.
   - There are no checks for the address(0) in updateTimelock function. Besides, the assignment to a newTimelock is done in one step, which means that if this function is called with the wrong parameter, access to all the governance functionality gets lost.
   - Updating the _timelock should be disabled since the TLC variable in GBL is not modifiable. If the _timelock gets changed in the Governance contract and not in GBL, all the functions restricted to onlyGovernance in ZivoeTranches would become inaccessible since the onlyGovernance checks that _msgSender() == GBL.TLC(), which means that they can not be called by the newTimeLock.

```
modifier onlyGovernance() {
    require(_msgSender() == IZivoeGlobals_ZivoeTranches(GBL).TLC(),
        "ZivoeTranches::onlyGovernance()        _msgSender()        !=
IZivoeGlobals_ZivoeTranches(GBL).TLC()");
    _;
}
```

2. An address, and only an address, with EXECUTOR_ROLE should be able to execute a proposal.
   - Only an address with EXECUTOR_ROLE can execute a proposal because the functions execute and executeBatch guarantee this property through the modifier:

```
modifier onlyRoleOrOpenRole(bytes32 role) {
    if (!hasRole(role, address(0))) {
        _checkRole(role, _msgSender());
```

```
    }
    _;
}
```

- According to the deployment scheme - function `deployCore` in `Utility.sol` - `address(0)` will have `EXECUTOR_ROLE`, which means that the execution of proposals is made public. Notice that it is possible, through a governance proposal, to grant the `EXECUTOR_ROLE` to another address.

- It is also possible through governance proposals to revoke the `EXECUTOR_ROLE` of `address(0)`, making execution of proposals forever unavailable if no other address was granted the `EXECUTOR_ROLE`.

3. A proposal should be executable if, and only if, the voting period has finished, a quorum is reached, and the vote succeeded.

   - A proposal is executable only if the voting period has finished, a quorum is reached, and the vote succeeded. The execute function in `Governor` requires that the proposal's `state` is `Succeeded` or `Queued`. The `state` function is overridden in `ZivoeGTC` and calls `super.state`, requiring that it returns `Succeeded`. The `super.state` only returns `Succeeded` if these 3 conditions are met.

   - However, a proposal may not be executable if the voting period has finished, a quorum is reached, and the vote succeeded. In order to be executable, a proposal must be queued and scheduled in the `_timelock`, which implies property 5. to hold. To be able to execute the proposal, property 2. must also hold.

4. A user, and only a user, with voting power greater than `proposalThreshold` can make a valid proposal.

   - True. This is guaranteed by the `propose` function in `Governor.sol`.

5. This contract should have `PROPOSER_ROLE` in `ZivoeTLC`.

   - After a proposal is made and succeeded, it should be queued and scheduled in `TLC` (see property 6.). The `queue` function (in `ZivoeGTC`) calls the `scheduleBatch` function (in `ZivoeTLC`). The `scheduleBatch` function has the modifier `onlyRole(PROPOSER_ROLE)`, which means that a succeeded proposal is only executable if `ZivoeGovernorV2` has `PROPOSER_ROLE` in `ZivoeTLC`.

   - In the deployment scheme, the governance contract is being granted `PROPOSER_ROLE`. However, it is possible, through governance proposals, to revoke or renounce `PROPOSER_ROLE`, making scheduling of proposals and consequently execution of proposals forever unavailable if no other address was granted the `PROPOSER_ROLE`.

6. A proposal can be executed or canceled if, and only if, it has been queued and scheduled in `TLC`.

- A proposal can be executed or canceled if it has been queued and scheduled in `TLC` only if property 2. holds. Notice that it is possible that the `TLC` gets without an `EXECUTOR_ROLE`, making the execution of proposals impossible to happen.
- A proposal can be executed or canceled only if it has been queued and scheduled in `TLC` because before executing a proposal, the `execute` function in `TLC` calls `_beforeCall` or `_beforeCallKeeper`. Both functions require that the operation is ready, through `isOperationReady` and `isOperationReadyKeeper`, respectively. The operation is only ready if the `timestamp` of the proposal is greater than `1` (`DONE_TIMESTAMP`). For this to be true, the proposal must have been scheduled before - the `schedule` function assigns a `timestamp` (`block.timestamp + delay`) to the proposal. The function can only be scheduled by an address with `PROPOSER_ROLE`. According to the deployment scheme, the governance contract is the only one with `PROPOSER_ROLE`. Therefore, a proposal can only be scheduled by queueing that proposal. Notice it is possible, through governance proposals, to grant the `PROPOSER_ROLE` to another address, and in that case, it may be possible to have a scheduled proposal without being queued.

7. After being scheduled, an address that is not a `keeper` can `execute` a proposal after some `delay` has passed. A `keeper` can `execute` a proposal if a `delay` minus 12 hours has passed.
   - True. Before executing a proposal, the `execute` function in `TLC` calls `_beforeCall` or `_beforeCallKeeper`. Both functions require `isOperationReady` or `isOperationReadyKeeper`, respectively. The operation is only ready if the current time point is greater than the `timestamp` of the proposal or greater than the `timestamp` of the proposal minus 12 hours for a `keeper`. The `timestamp` of the proposal is assigned in the `schedule` function and is equal to the `block.timestamp` plus the `delay`.
8. The voting weight of a user is the sum of his `ZVE` balance plus the amount of his staked `ZVE` in `stZVE` and `vestZVE`.
   - True. Function `_getVotes` is being overridden and returns:

```
token.getPastVotes(account, blockNumber) +
GBL.vestZVE().balanceOf(account) + GBL.stZVE().balanceOf(account);
```

9. Governance, and only governance, should be able to change parameters such as `_votingDelay`, `_votingPeriod`, `_proposalThreshold`, `_quorumNumerator`, `minDelay` and addresses with `PROPOSER_ROLE`, `EXECUTOR_ROLE` and `CANCELER_ROLE`.
   - The setters for these parameters, such as `setVotingDelay`, `setVotingPeriod` and `setProposalThreshold` in `GovernorSettings`, `updateQuorumNumerator` in `GovernorVotesQuorumFraction` and `updateDelay` in `ZivoeTLC`, have the modifier `onlyGovernance`, which requires `_msgSender() == _executor()`. The function `_executor` is being overridden and returns the `_timelock`, which means that

the `msg.sender` should be `_timelock`. The `_timelock` can only execute valid proposals (see property 3), which means that ultimately only governance is able to change these parameters.

- If properties 2. 3. and 4. are holding, governance is able to change these parameters. However, notice that being able to change these parameters can be risky. If a governance proposal goes through and reaches a quorum to change these parameters to unreasonable values, the protocol gets inoperable.

ZivoeITO.sol

1. It should be possible to deposit allowed stablecoins, and only allowed stablecoins, to this contract.

    - True. Allowed stablecoins can be deposited through `depositJunior` or `depositSenior` functions. Both functions require the `asset` being deposited to be an allowed stablecoin:

    ```
    require( asset == stables[0] || asset == stables[1]
            || asset == stables[2] || asset == stables[3],
            "ZivoeITO::depositJunior() asset != stables[0-4]"
    );
    ```

2. The `ITO` period should be a reasonable amount of time.

    - According to the deployment scheme - `deployCore` - the `ITO` will be 30 days. However, there's no enforcement in the contract for the maximum duration of the `ITO`.

3. Depositors to Junior Tanche should receive `1 pZVE` and `1 zJTT` per stablecoin deposited.

    - True. In the function `depositJunior`, the depositor gets as many credits as the standardized amount of the stablecoin he deposited:

    ```
    juniorCredits[caller] += standardizedAmount;
    ```

    - When claiming the `depositor` receives the exact amount of credits of `zJTT` and `pZVE`, the latest is converted to the right proportion of `ZVE`.

4. Depositors to Senior Tranche should receive `3 pZVE` and `1 zSTT` per stablecoin deposited.

    - True. In the function `depositSenior`, the depositor gets 3 times the credits the standardized amount of the stablecoins he deposited:

    ```
    seniorCredits[caller] += standardizedAmount;
    ```

    - When claiming the `depositor` receives the exact amount of credits of `pZVE`, which is converted to the right proportion of `ZVE`. And receives ⅓ of the amount of credits he owns of `zSTT`.

5. Users can claim their tokens and start their `vestingSchedule` after, and only after, ITO concludes.
    - True. The function `claimAirDrop` requires `block.timestamp > end || migrated`.
6. After `ITO` ends, it should not be possible to make any deposits to this contract.
    - True. Both functions, `depositJunior` and `depositSenior` require `!migrated`. The `migrated` flag is only set to `true` when `ITO` ends and never changes its value again.
7. After ITO ends, the funds deposited to the accepted stable coins should be transferred to `DAO`.
    - True. In the function `migrateDeposits`, responsible for finishing `ITO`, each stablecoin's balance is completely transferred to `DAO`.
8. After, and only after, ITO ends, the `YDL` and `ZVT` contracts should be `unlocked`.
    - True. The `YDL` and `ZVT` contracts only get `unlocked` in the function `migrateDeposits`, which is responsible for finishing `ITO`.
9. `ZVL`, and `ZVL` only, should be able to close ITO at any time. Migration can be triggered by anyone after the `ITO` ends.
    - True. The function `migrateDeposits` only requires that `block.timestamp > end` if the `msg.sender` is different than `ZVL`, which means that `ZVL`, and `ZVL` only, can close `ITO` at any time, whereas anyone else can trigger migration after `ITO` ends.

```
if (_msgSender() != IZivoeGlobals_ITO(GBL).ZVL()) {
    require(block.timestamp > end,
       "ZivoeITO::migrateDeposits() block.timestamp <= end");
}
```

10. This contract should have minting privileges for the `zJTT` and `zSTT` contracts.
    - According to the deployment scheme - `deployCore` in `Utility.sol` - the `owner` of `zJTT` and `zSTT` (the deployer) grants `minter` role to `ITO`.
11. After `ITO` ends, this contract should not be able to mint `zJTT` and `zSTT`.
    - This contract only mints `zJTT` and `zSTT` in the functions `depositJunior` and `depositSenior`, respectively. After `ITO` ends, according to property 6., it is not possible to call these functions, therefore this contract does not mint `zJTT` and `zSTT` after `ITO` ends.
12. Right after `ITO`, `zJTT.totalsupply` should be equal to the sum of all deposits in Junior Tranche and `zSTT.totalsupply` should be equal to the sum of all deposits in Senior Tranche.

- According to the deployment scheme, the only contracts with minting privileges on `zJTT` and `zSTT` are `ITO` and `ZVT`. To `mint zJTT` or `zSTT`, the `ZVT` contract should be `unlocked`. According to property 8., the `ZVT` contract only gets `unlocked` after `ITO` ends. According to properties 3. and 4., `ITO` mints `1 zJTT`, or `1 zSTT`, per stablecoin deposited, which means that `zJTT.totalSupply` is equal to the sum of all deposits in Junior Tranche and `zSTT.totalsupply` is equal to the sum of all deposits in Senior Tranche.

13. Funds deposited during `ITO` can be migrated only once.

    - True. The function migrateDeposits requires !migrated. The `migrated` flag is set to `true` after this check and never changes its value again.

ZivoeMath.sol

1. Calculated EMA (exponential moving average) should be in the range `[min(bV,cV), max(bV,cV)]`

   The formula used for ema calculation is

   $eV = \frac{(M*cV)+(WAD-M)*bV}{WAD}$ Re-organizing this formula gives us

   $$eV = \frac{(M*cV)+WAD*bV-M*bV}{WAD}$$

   $$eV = \frac{M*(cV-bV)}{WAD} + \frac{WAD*bV}{WAD}$$

   $eV = \frac{M*(cV-bV)}{WAD} + bV$ replacing M where $N > 0$

   $$eV = \frac{\frac{WAD*2}{N+1}*(cV-bV)}{WAD} + bV$$

   $$eV = \frac{WAD*2*(cV-bV)}{(N+1)*WAD} + bV$$

   $$eV = \frac{2*(cV-bV)}{(N+1)} + bV$$

   $$eV = \frac{2}{(N+1)}*(cV - bV) + bV$$

   The first term in the addition specifies how much eV deviates from bV.

   The largest value for the first term in the addition is when N=1 which is as follows

$eV = cV - bV + bV$

$eV = cV$

The smallest value is 0 due to rounding in the integer division which gives
$eV = bV$

2. Exponential moving average should not be calculated for `n≤0`

`ema()` function is only used in `ZivoeYDL::distributeYield()` where parameter N is used as `min(retrospectiveDistributions, distributionCounter)` where `retrospectiveDistributions` is set constantly to value 6 making the range of N as `[distributionCounter, 6]`. Moreover, `distributionCounter` is immediately incremented before `ema()` calls making its minimum value 1 so the actual range of N becomes `[1,6]`

3. Proportion of yield attributable to junior tranche is 0 when proportion of yield attributable to seniors is greater than or equal to 1

$jP=(...).min(RAY-sP)$ The first part of the calculation is omitted since it is irrelevant for the proof

$jP=(...).min(RAY-RAY)$

$jP=(...).min(0)$ which is 0.

4. Precision of calculated proportion of yield attributable to junior tranche should be in RAY

For all the values where $sP \geq RAY$ function takes 0 value. For the rest, following formula is used

$jP=(...).min(RAY-sP)$ where $sP=0$ assigns maximum value to the formula

$jP=(...).min(RAY-sP)$ hence the maximum value of $jP=RA$

The precision analysis of the formula being used is as follows:

$$jP = \frac{eJTT * sP * \frac{Q}{BIPS}}{eSTT}$$ re-organizing this formula as

$$jP = sP * \frac{eJTT}{eSTT} * \frac{Q}{BIPS}$$ current constant value of $Q=16250$

$$jP \ = \ sP * \frac{eJTT}{eSTT} * \frac{16250}{10000}$$

`jP` will be in `RAY` since `sP` is in `RAY` and `eJTT`,`eSTT` couple is both in `WEI`.

For the actual solidity code, an additional range analysis might be useful. For the used formula

$$jP \ = \ \frac{\frac{Q*eJTT*sP}{BIPS}}{eSTT}$$

initially the multiplication `Q*eJTT*sP` is executed where the result is in range `BIPS*WEI*RAY`

$$jP \ = \ \frac{\frac{BIPS*WEI*RAY}{BIPS}}{eSTT}$$

$$jP \ = \ \frac{WEI*RAY}{eSTT}$$ where eSTT is expected to be in range `WEI`

$$jP \ = \ \frac{WEI}{WEI} * RAY$$

making jP in `RAY` precision

5. Precision of calculated proportion of yield attributable to senior tranche should be in `RAY`

   `sP` is calculated with a piecewise function where `seniorProportionShortfall()` is used if the distributable yield is smaller than the target yield and `seniorProportionBase()` is used otherwise.

   For both of the functions formulas of the following form is used

   sP=(...).floorDiv(...).min(RAY) where a minimum value of 0 can be returned due to `floorDiv()` and a maximum value of `RAY` can be returned due to `min(RAY)` invocations.

   The precision analysis of the formula being used is as follows.

   For `seniorProportionShortfall()` function:

$$sP \ = \ \frac{WAD*RAY}{WAD + \frac{Q*eJTT*\frac{WAD}{BIPS}}{eSTT}}$$ re-organizing this formula as

$$sP \ = \ \frac{WAD*RAY}{WAD*eSTT + WAD*\frac{Q*eJTT}{BIPS}}$$

$$sP = \frac{RAY}{eSTT + \frac{Q*eJTT}{BIPS}}$$

$$sP = \frac{RAY}{\frac{eSTT*BIPS + Q*eJTT}{BIPS}}$$

$$sP = \frac{RAY*BIPS}{eSTT*BIPS + Q*eJTT}$$ current constant value of Q=16250

$$sP = RAY*\frac{10000}{eSTT*10000+16250*eJTT}$$

For the final formula both eSTT and eJTT are in range `WEI` making the range of the function in `RAY` precision.

For the actual solidity code, an additional range analysis might be useful. For the used formula

$$sP = \frac{WAD*RAY}{WAD + \frac{\frac{Q*eJTT*WAD}{BIPS}}{eSTT}}$$

initially the multiplication `Q*eJTT*WAD` is executed where the result is in range `BIPS*WEI*WAD`

$$sP = \frac{WAD*RAY}{WAD + \frac{\frac{BIPS*WEI*WAD}{BIPS}}{eSTT}}$$

$$sP = \frac{WAD*RAY}{WAD + \frac{WEI*WAD}{WEI}}$$ where eSTT is expected to be in range `WEI`

$$sP = \frac{WAD*RAY}{WAD+WAD}$$

$$sP = \frac{WAD}{WAD}*RAY$$

making sP in `RAY` precision

For `seniorProportionBase()` function:

$$sP = \frac{\frac{RAY*Y*eSTT*\frac{T}{BIPS}}{365}}{yD}$$ re-organizing this formula as

$$sP \;=\; \frac{RAY*Y*eSTT*T}{BIPS*365*yD}$$ current constant value of T=30 and Y=800

$$sP \;=\; \frac{RAY*800*eSTT*30}{10000*365*yD}$$

$$sP \;=\; RAY*\frac{eSTT}{yD}*\frac{24}{3650}$$

For the final formula both eSTT and yD are in range `WEI`, making sP in `RAY` precision.

For the actual solidity code, an additional range analysis might be useful. For the used formula

$$sP \;=\; \frac{\frac{\frac{RAY*Y*eSTT*T}{BIPS}}{365}}{yD}$$ initially the multiplication `RAY*Y*eSTT*T` is executed where the result is in range `RAY*BIPS*WEI*T`

$$sP \;=\; \frac{\frac{\frac{RAY*BIPS*WEI*T}{BIPS}}{365}}{yD}$$

$$sP \;=\; \frac{\frac{RAY*WEI*T}{365}}{yD}$$ assuming T=30 and yD is in range `WEI`

$$sP \;=\; \frac{RAY*WEI}{WEI} \;=\; RAY*\frac{WEI}{WEI}$$

making sP in `RAY` precision.

6. Precision of calculated annual yield required to meet target rate should be in `WEI`

   The formula used in calculation is as follows:

$$yT \;=\; \frac{Y*T*(eSTT+eJTT*\frac{Q}{BIPS})}{BIPS*365}$$ reorganizing the formula as

$$yT \;=\; \frac{T}{365}*\frac{Y*(eSTT+eJTT*\frac{Q}{BIPS})}{BIPS}$$ current constant value of T=30 and Y=800

$$yT \;=\; \frac{30}{365}*\frac{Y}{BIPS}*(eSTT \;+\; eJTT*\frac{Q}{BIPS})$$ current constant value of Q=16250

$$yT \;=\frac{30}{365}*\frac{800}{10000}*eSTT \;+\; \frac{30}{365}*\frac{800}{10000}*\frac{16250}{10000}*eJTT*\frac{Q}{BIS}$$

For the final formula both eSTT and eJTT is in range `WEI`, thus the returned result is in `WEI` precision.

For the actual solidity code, an additional range analysis might be useful. For the used formula

$$yT = \frac{\frac{Y*T*(eSTT+\frac{eJTT*Q}{BIPS})}{BIPS}}{365}$$

initially the multiplication eJTT*Q is executed where the result is in range `WEI*BIPS`

$$yT = \frac{\frac{Y*T*(eSTT+\frac{WEI*BIPS}{BIPS})}{BIPS}}{365}$$

$$yT = \frac{\frac{Y*T*(eSTT+WEI)}{BIPS}}{365}$$

where eSTT is expected to be in range `WEI` and Y is expected to be in range `BIPS`

$$yT = \frac{\frac{BIPS*T*(WEI+WEI)}{BIPS}}{365}$$

$$yT = \frac{T*(WEI+WEI)}{365}$$ assuming T=30

$$yT = \frac{30}{365}*(WEI + WEI)$$

Thus the returned result is in `WEI` precision.

Summary of precision analysis:

- Calculated EMA is in the range `[min(bV,cV), max(bV,cV)]`
- Precision of calculated proportion of yield attributable to junior tranche is in `RAY`
- Precision of calculated proportion of yield attributable to senior tranche is in `RAY`
- Precision of calculated annual yield required to meet target rate is in `WEI`

ZivoeRewards.sol

1. There may be at most 10 reward tokens present.
    - The list of tokens to be used in vesting rewards are kept in `rewardTokens` array. `ZVL` account can use `addReward` function to push a token address to this array. Before the push operation it is ensured that the length of the array is less than 10 which allows at most 10 token addresses to be present in the array. In its current status of the code, it is not possible to remove a token address from the rewards array.
2. A reward token shall not be added twice.
    - In the `addReward` function, the `rewardsDuration` property of the specific token address in the `rewardData` mapping is set to a non-zero value. It is not possible to set

this value in any other way, so any token pushed to `rewardTokens` array has its `rewardsDuration` property set to a non-zero value. Before a token is pushed to the array it is ensured that this property is zero, hence it is not already in the `rewardTokens` array.

3. Last point in time where reward may be applicable should always be later than the last time of the update.

   ○ `periodFinish` variable in the `Reward` data structure keeps the final point in time where a reward is applicable. This variable is updated (incremented, pushed forward as `timestamp`) every time a `depositReward` is invoked for the corresponding reward token.

   ○ `lastUpdateTime` variable in the `Reward` data structure keeps the last time that the reward value for the token is updated. During a `depositReward` invocation this value is updated to the current block's timestamp where `periodFinish` is updated with a larger value as described above.

   ○ The second point where `lastUpdateTime` is updated is in the `updateReward` modifier where the value is updated to the smaller value of the current block's timestamp and `periodFinish`.

   ○ Finally, the initial value of `lastUpdateTime` is zero which may affect the correctness of `updateReward` modifier since it has actively been used in `rewardPerToken` calculation. However this calculation also uses `rewardRate` which is initially zero, making the `earned` rewards zero. `rewardRate` is initialized in `depositReward` function which updates `lastUpdateTime` to `block.timestamp`.

4. The `rewardsDuration` of any `_rewardToken` should be a reasonable amount of time.

   ○ During the addition of each reward token the `rewardsDuration` variable is checked to be larger than 0. However there's no upper limit check for the duration. This variable can be assigned with a very large value to practically yield 0 rewards.

5. The `_totalSupply` is equal to the sum of all staked amounts minus the sum of all withdrawn amounts.

   ○ There exists three points of `_totalSupply` modification:

      ● `stake()` and `stakeFor()` functions where the amount to be staked is added to `_totalSupply`

      ● `withdraw()` function where the amount to be withdrawn is subtracted from `_totalSupply`

   Since there are no other points of modification, `_totalSupply` variable keeps the stated property properly

6. The `balanceOf` a given `account` is equal to the sum of all staked amounts (through `stake` or `stakeFor`) for the `account` minus the sum of all withdraws made by the `account`.
   - LP token balances are kept in `_balances` mapping. There exists three points of balance modification for an account
     - `stake()` and `stakeFor()` functions where the amount to be staked is added
     - `withdraw()` function where the amount to be withdrawn is subtracted

   Since there are no other points of modification, `_balances` variable keeps the stated property properly

7. For any `_rewardToken`, the `rewardData[_rewardToken].rewardPerTokenStored` is less or equal than the total amount of `depositReward` for that `_rewardToken`.
   - `rewardPerTokenStored` is only updated by the `updateReward` function. Initially this value is 0 and it is being calculated zero by `updateReward` as long as there are no deposits. This is due to the `rewardRate` being zero before any reward deposits. `rewardRate` is being used as a multiplicator in the updateReward function.
     Only during a reward deposit `rewardRate` is updated. If the reward distribution period is due it is updated to $\frac{amount}{duration}$
     Otherwise, if there is an ongoing distribution process it is updated to $\frac{amount+leftover}{duration}$
     where leftover is the amount left to be distributed in the current reward distribution.
     In summary, the `rewardRate` is the current amount of rewards to be distributed divided by total duration of reward distribution.

   - In the rewardPerToken function where the update value of `rewardPerTokenStored` is being calculated, the calculation updates the value for non-zero `_totalSupply` value as follows. Current value of `rewardPerTokenStored` is incremented with $\frac{\Delta_t * rewardRate*WAD}{\_totalSupply}$ which in turn is $\frac{\Delta_t * reward*WAD}{duration*\_totalSupply}$ taking out the precision factor (WAD) and re-organizing the equation gives us: $reward * \frac{\Delta_t}{duration*\_totalSupply}$
     This formula guarantees that the `rewardPerToken` is less than the reward being distributed since property 3 guarantees $\Delta_t \leq dur$

8. For any `token` and any `account`, the `accountRewardPerTokenPaid[account][token]` is less or equal than `rewardData[token].rewardPerTokenStored`.

- `accountRewardPerTokenPaid` keeps the previous value of `rewardPerTokenStored` before the `earned` function call. When `updateReward` is called with an address parameter other than `address(0)` this value is updated with `rewardPerTokenStored`. This implies a non-decreasing value of `rewardPerTokenStored`. In `rewardPerToken` function the increase calculated is greater than or equal to 0 so this property holds.

9. For any `_rewardToken`, the `rewardData[_rewardToken].lastUpdateTime` should be updated every time there is a deposit of `_rewardToken`, a user withdraws his `stakingTokens`, a user stakes `stakingTokens` to the contract, a user claims his rewards.
   - For all the listed actions the modifier `updateReward` is called, updating the value.
10. At any second, for any `_rewardToken`, the amount of rewards that an `account` gets rewarded is equal to: $balances[account] \ * \ \frac{rewardData[\_rewardsToken].rewardRate}{\_totalSupply}$
    - For each account and rewardToken, the total claimable awards are updated within `updateRewards` modifier (which is applied at each critical operation). In this modifier the total amount of rewards collectible for an account is incremented by the accounting of the `earned` function. In this function the amount of incrementation is calculated by multiplying $balances[account]$ with:

$$\frac{rewardsPerToken(\_rewardsToken) - accountRewardPerTokenPaid[account][\_rewardsToken]}{WAD}$$

In this formula, when `earned` is called within `updateReward` modifier, `rewardData[_rewardsToken].lastUpdateTime` is updated to the minimum value of `block.timestamp` and `periodFinish` time by `lastTimeRewardApplicable` function. Afterwards, during `earned`, `lastTimeRewardApplicable` function is called one more time during `rewardPerToken` function and subtracts `rewardData[_rewardsToken].lastUpdateTime` from the value returned which results in 0. This in turn results `rewardPerToken` return `rewardData[_rewardToken].rewardPerTokenStored` (no increase) all the time regardless of `_totalSupply` being 0 or not.

As a result the multiplication of $balances[account]$ will be done with:

$$\frac{\_rewardToken.rewardPerTokenStored - accountRewardPerTokenPaid[account][\_rewardToken]}{WAD}$$

Taking out the precision factor (WAD) this formula basically calculates the difference between reward rates between two points in time and returns the rate applied to `balances[account]` as the earned reward. Since the rate is guaranteed to be non-decreasing this calculation returns the non-applied difference in rates.

11. The owner, and only the owner, of some `account` should be able to claim the rewards of that `account`.
    - The only functions to claim the rewards are `getRewards` and `getRewardAt` which both use `_msgSender` during updates and transfers.
12. For each `_rewardToken`, an `account` should be able to claim the `amount`, and only the `amount`, of rewards earned by the `account`.
    - For each reward claimed, only the full amount of rewards can be withdrawn. This is ensured by using `_msgSender` in accessing and resetting the account reward in the `rewards` mapping.
13. An `account` should be able to withdraw any `amount` between 0 and `balanceOf(account)` of the `stakingToken`.
    - The only function to withdraw the staked amount is the `withdraw` function which ensures an amount greater than 0 is supplied. It also ensures by the subtraction operations that `amount ≤ _balances[_msgSender()]` and `amount ≤ _totalSupply`
14. It should only be possible to deposit reward after the related reward token is added.
    - The `.div(rewardData[_rewardsToken].rewardsDuration` operation(s) in the `depositReward` function would revert since it is guaranteed that this variable is set to some value other than 0 during `addReward` function.

ZivoeRewardsVesting.sol

1. There may be at most 10 reward tokens present.
    - The list of tokens to be used in vesting rewards are kept in `rewardTokens` array. ZVL account can use `addReward` function to push a token address to this array. Before the push operation it is ensured that the length of the array is less than 10 which allows at most 10 token addresses to be present in the array. In its current status of the code, it is not possible to remove a token address from the rewards array.
2. A reward token shall not be added twice.
    - In the `addReward` function, the `rewardsDuration` property of the specific token address in the `rewardData` mapping is set to a non-zero value. It is not possible to set this value in any other way, so any token pushed to `rewardTokens` array has its `rewardsDuration` property set to a non-zero value. Before a token is pushed to the array it is ensured that this property is zero, hence it is not already in the `rewardTokens` array.
3. There may only be a single vesting schedule per account.

- `vestingScheduleSet` mapping keeps a logical `true` value if the vesting schedule for the account is set. Within the `createVestingSchedule` function it is ensured that this value for the account is set to `false` and then it is set to `true` immediately after necessary checks. It is not possible to set a vesting schedule for an account in any other way.

4. If revocable, a vesting schedule can be revoked at most once.

    - Schedule revocability is kept by `revokable` property of the `VestingSchedule` data structure. It is only possible to revoke a vesting schedule by the invocation of `revokeVestingSchedule` function which ensures that this property is set to `true`. Once successfully revoked, this value is set to `false`. Together with the property of being able to schedule a vesting for an account only once, this ensures a vesting schedule can be revoked only once.

5. At any time, the total amount of withdrawn assets shall not exceed the total vesting amount for the asset.

    - `withdraw` function uses `amountWithdrawable` to determine the amount to be withdrawn. `amountWithdrawable` only returns a value other than 0 if the current block's timestamp is after the cliff time. If the time hasn't reached the end of the schedule, amount to be withdrawn is determined as

      $(\Delta time * vestingRate) - totalWithdrawn$.

      If the schedule is past due `totalVesting - totalWithdrawn` is returned.

      Later in the `withdraw` function `totalWithdrawn` value is accumulated with the amount returned. Throughout the contract, `totalWithdrawn` can only be updated by the return value of `amountWithdrawable` function.

      Assume there are $n + 1$ invocations of `amountWithdrawable`, $n + 1$ being the last call after the end of the schedule. At any point in time where invocation $k \le n + 1$ is about to happen, $totalWithdrawn = \sum_{i=1}^{k-1} amount_i$, and `amountWithdrawable` will return:

      $(\Delta time_{1..k} * rate) - totalWithdrawn$

      $(\Delta time_{1..k} * rate) \ge totalWithdrawn$

      Because $\Delta time$ increases due to the calculation `block.timestamp-vestingStart.`

      The largest value to be calculated for invocations up to k would be: $vestingEnd - vestingStart$, which in turn makes the maximum value to be withdrawn as $\Delta vesting * vestingPerSecond$. Since `vestingPerSecond` is calculated as `amountToVest / (daysToVest * 1 days)` this value would be smaller than or equal to `amountToVest` (due to rounding down).

In the invocation of `amountWithdrawable` for the $k + 1$ time the residual would be withdrawn since it is calculated as `totalVesting-totalWithdrawn` (`totalVesting` is equal to `amountToVest`)

6. Reward token to be vested shall not be ZVE.

    ○ This is ensured by a check in the `addReward` function which is the only possible way to add a reward token.

7. Last point in time where reward may be applicable should always be later than the last time of the update.

    ○ `periodFinish` variable in the `Reward` data structure keeps the final point in time where a reward is applicable. This variable is updated (incremented, pushed forward as `timestamp`) every time a `depositReward` is invoked for the corresponding reward token.

    ○ `lastUpdateTime` variable in the `Reward` data structure keeps the last time that the reward value for the token is updated. During a `depositReward` invocation this value is updated to the current block's timestamp where `periodFinish` is updated with a larger value as described above.

    The second point where `lastUpdateTime` is updated is in the `updateReward` modifier where the value is updated to the smaller value of the current block's timestamp and `periodFinish`.

    ○ Finally, the initial value of `lastUpdateTime` is zero which may affect the correctness of `updateReward` modifier since it has actively been used in `rewardPerToken` calculation. However this calculation also uses `rewardRate` which is initially zero, making the `earned` rewards zero. `rewardRate` is initialized in `depositReward` function which updates `lastUpdateTime` to `block.timestamp`.

8. It should only be possible to deposit reward after the related reward token is added.

    ○ The `.div(rewardData[_rewardsToken].rewardsDuration` operation(s) in the `depositReward` function would revert since it is guaranteed that this variable is set to some value other than 0 during `addReward` function.

ZivoeToken.sol

1. ZVE should have an initial supply of 25M Ethers, which may only decrease.

   True. Constructor mints an initial supply of 25M Ethers. In its current condition `mint()` and `burn()` functions are not invoked by the protocol.

ZivoeTrancheToken.sol

1. Only TT (Tranche Token) minters that are designated by the governance can mint
   - True. Ensured by the `changeMinterRole()` function and the `isMinterRole()` modifier applied to overridden `_mint()` function
2. TT owner can burn its tokens
   - True. Ensured by the overridden `burn()` function.

ZivoeTranches.sol

1. Once the tranches are unlocked, if the total supply of JTT exceeds a ratio of STT designated by the variable `maxTrancheRatio` deposits to the junior tranche should be blocked.
   - `isJuniorOpen` function checks if an amount to be deposited to junior tranche causes an overflow in the ratio. `depositJunior` ensures the check by calling `isJuniorOpen` for every deposit.
2. `maxTrancheRatio` should be between 0% and 45%
   - True. `maxTrancheRatioBIPS` is initially set to 42.5% and the `updateMaxTrancheRatio` function ensures that this value does not overflow 45%. The value itself is `uint256` ensuring a lower bound of 0%
3. Awarded amount of ZVE tokens for a deposit to junior tranche should not exceed the balance of ZivoeTranches
   - True. Ensured by a check at the end of `rewardZVEJuniorDeposit` function
4. Awarded amount of ZVE tokens for a deposit to senior tranche should not exceed the balance of ZivoeTranches
   - True. Ensured by a check at the end of `rewardZVESeniorDeposit` function
5. The incentives ratio limits must be between 10% and 25%.
   - True. Initially this range is set to [10%, 20%] range. Lower and upper bounds can be updated via function. Lower and upper bound updates perform necessary upper and lower bound checks.

6. The upper limit of the incentives ratio should be strictly greater than the lower limit of the incentives ratio
    - Lower bound update performs the necessary check (<upper). However upper bound update functions do not check if the upper>lower is kept by the update.
7. The limits of the minimum amount of ZVE tokens minted per JTT tokens should be between 0 and 0.1 ETH.
    - True. `minZVEPerJTTMint` and `maxZVEPerJTTMint` variables keep track of these ratios. Initially both these ratios are set to 0 and later updating functions performs the necessary checks.
8. The upper limit of the ZVE tokens minted per JTT tokens should be strictly greater than the lower limit.
    - Initially both of these limits are set to 0, violating this property since values are equal.
    - Updates to the upper limits do not control upper>lower check which may potentially violate this property.

ZivoeYDL.sol

1. YDL starts yield distribution at least 30 days later than the time it has been unlocked.
    - True. `distributeYield` function contains a check where the current block's timestamp should be at least `daysBetweenDistributions * 86400` greater than `lastDistribution`. Here, `daysBetweenDistributions` is set constantly to 30 and 86400 is the number of seconds in one day.
    - `lastDistribution` is initially set to 30 days later than the time `unlock` is called for the contract. `unlock` can be called only once since the `migrateDeposits` function can be called only once (see `ZivoeITO` properties) hence `lastDistribution` can be initialized only once.
2. `protocolEarningsRate` should be between 0% and 90%.
    - True. `protocolEarningsRateBIPS` is declared as `uint256` and initialized to 2000 at variable declaration. `updateProtocolEarningsRateBIPS()` function requires the new value to be less than or equal to 9000.
3. The proportion of yield distribution for each recipient should be greater than 0.
    - True. Yield distribution proportions are calculated in `earningsTrancheuse` function and returned in `protocol` and `residual` arrays separately. Additionally, the amount of

yield to be distributed to senior and junior tranches are calculated in this function where each proportion is calculated separately.

- Both protocol and residual recipient proportion values in BIPS are set in `updateRecipients` function. In this function, it is checked if each proportion is greater than 0 and all proportions add up to `BIPS` (100%)

4. The actual amount of total distributed yield should be smaller than or equal to the `earnings.`

   - True. `earnings` are distributed in four parts as
     - `protocolEarnings (yP)`
     - `postFeeYield (yD)`
       - `residualEarnings (yDR)`
       - `seniorTrancheEarnings (yDS)`
       - `juniorTrancheEarnings (yDJ)`

   This break-up is guaranteed to satisfy the following:

   - `earnings = yP+yD`
   - `yDR = yD - (yDS+yDJ)`

   Hence we break this proof down to four parts:

   a. The actual amount of yield distributed to protocol should be smaller than or equal to the calculated `protocolEarnings`.

      For `protocolEarnings`, `earningsTrancheuse` function iterates through the proportions of the recipients where proportion set is guaranteed to add up to 100% in the `updateRecipients` function. Hence during the iterations

      $\sum_{1}^{rec.length} \frac{prop_i * yP}{BIPS}$ would equal to: $(\sum_{1}^{rec.length} prop_i) * \frac{yP}{BIPS}$, which is $BIPS * \frac{yP}{BIPS} = yP$. However, due to the rounding down during integer division $\sum_{1}^{rec.length} \frac{prop_i * yP}{BIPS} \leq yP$.

   b. The actual amount of yield distributed as residual should be smaller than or equal to the calculated `residualEarnings`. For `residualEarnings`, `earningsTrancheuse` function iterates through the proportions of the recipients where proportion set is guaranteed to add up to 100% in the `updateRecipients` function. Hence, during the iterations $\sum_{1}^{rec.length} \frac{prop_i * yD_R}{BIPS}$

would equal to $(\sum_{1}^{rec.length} prop_i) * \frac{yD_R}{BIPS}$, which is $BIPS * \frac{yD_R}{BIPS} = yD_R$. However, due to the rounding down during integer division $\sum_{1}^{rec.length} \frac{prop_i * yD_R}{BIPS} \leq yD_R$.

For both of yP and yD_R above (4.1 and 4.2), during distribution of assets, the distributed yield may further be broken down into two parts if the recipient is the rewards contract. This distribution is performed following the ratios $\frac{bal(stZVE)}{bal(stZVE)+bal(vestZVE)}$ and $\frac{bal(vestZVE)}{bal(stZVE)+bal(vestZVE)}$ accordingly where each ratio is ≤ 1. This distribution also guarantees less than or equal amount to be distributed. Hence, yP and yD_R both are guaranteed to be less than or equal to their actual values.

c. The actual amount of yield distributed to senior Tranche should be smaller than or equal to the calculated `seniorTrancheEarnings`.

   Yields are directly distributed to the senior tranche as calculated by the `ZivoeMath::seniorProportion()` in the `earningsTrancheuse()` function. `ZivoeMath::seniorProportion()` is a bipartite function where either `seniorProportionShortfall()` or `seniorProportionBase()` is used for calculation. Both function guarantee to return a value in `[0, RAY]` and hence guarantee that the calculated senior yield would be $yDS \leq yD$.

d. The actual amount of yield distributed to junior Tranche should be smaller than or equal to the calculated `juniorTrancheEarnings`.

   Yields are directly distributed to the junior tranche as calculated by the `ZivoeMath::juniorProportion()` in the `earningsTrancheuse()` function. `ZivoeMath::juniorProportion()` guarantee to return a value in [0, RAY] and hence guarantee that the calculated junior yield would be $yDJ \leq yD$.

   This by itself is not sufficient for the $yDS + yDJ \leq yD$. However, this function not only guarantees to return a value in `[0, RAY]` but to return a value in an even smaller range in `[0, RAY - yDs]` making $yDJ \leq yD - yDS$ and hence making $yDS + yDJ \leq yDS + yD - yDS$ and $yDS + yDJ \leq yD$.

   The piecewise combination of 4.1- 4.4 proves the main property. It is guaranteed: $yDS + yDJ \leq yD$ and $yDR \leq yD - (yDS + yDJ)$, hence $yDS + yDJ + yDR \leq yD$.

Also, for calculated yP' (calculated yP): $yP' \leq yP$. Hence, $yD' \leq yD$ and $yP' \leq yP$ and $yD' + yP' \leq earnings$.

This proves it is not possible to distribute more than the actual earnings but it also shows that there may still be some residual yield after the distribution.

5. There should be at least 30 days between invocation of yield distributions; distributions should be calculated over exactly 30 day periods.

    - True. Every time `distributeYield` is called `lastDistribution` is updated to `block.timestamp` and later for each invocation it is required that the current block's timestamp is at least 30 days later than the `lastDistribution`. During the accounting of senior proportions and yield targets `daysBetweenDistributions` constant is used (as 30).

6. The adjusted amount of STT should be strictly greater than the adjusted amount of JTT unless both are 0.

    - True. Adjusted supplies are calculated by the `ZivoeGlobals::adjustedSupplies()` function $JTT_{asup} \leq JTT_{sup}$ since adjusted supply value is calculated by subtracting a positive value from the actual supply value.

      Afterwards, adjusted senior supply value is calculated piecewise. If (indirectly checked) $JTT_{asup} > 0$ then $STT_{asup} = STT_{sup}$ otherwise $STT_{asup} < STT_{sup}$.

      For the second case, since $JTT_{asup} = 0$, it is guaranteed that $JTT_{asup} < STT_{asup}$ unless $JTT_{asup} = 0$.

      For the first case, it is guaranteed that $JTT_{asup} \leq JTT_{sup}$, but the relation between JTT_{asup} and STT_{asup} depends on the relation between JTT_{sup} and STT_{sup}. However $JTT_{sup} < STT_{sup}$ property is ensured by `ZivoeTranches` hence adjusted supply values should also follow.

Findings

A01: `lockers` can transfer all `ERC721`

[Severity: High | Difficulty: High | Category: Security]

The functions `pushERC721` and `pushMultiERC721` are responsible for pushing NFT's from `DAO` to some `locker`. In order to accomplish that, these functions must give allowance to the `locker` to transfer the NFT, and call the respective `push` function in the locker. However, both functions call `IERC721(asset).setApprovalForAll(locker, true)`, giving the `locker` more privileges than what they should.

Scenario

1. A malicious `ERC721 locker` gets whitelisted through a governance proposal.
2. A proposal to transfer some `ERC721 tokenId`, or a list of `tokenIds`, gets approved.
3. The `locker` is able to transfer all the `ERC721 tokens` from `DAO`.

Recommendation

Replace the call to `setApprovalForAll(locker, true)` with `approve(locker, tokenId)` in the `pushERC721` function, and with `approve(locker, tokenIds[i])` in the function `pushMultiERC721`.

Status

Addressed in commit 7fae9e40edeb2b0875e358ee182ffc80f0e73fcf.

A02: Transfer of ZVL access control is unprotected

[Severity: High | Difficulty: High | Category: Input Validation]

A call to `ZivoeGlobals::transferZVL()` directly transfers ownership to the provided address, which grants a great extent of privileges to the transferred address. A significant amount of functionality is unusable in case of transfer to `address(0)` or to an address that the ZivoeLabs has no access to.

Scenario

One of the following scenarios is possible:

1. `ZVL` calls `transferZVL` unintendedly, giving as parameter `address(0)`.
2. `ZVL` pretends to transfer access control to another address but mistype the new address.

Both cases lead to a great loss of protocol functionality since many functions are restricted to `ZVL`.

Recommendation

A 2-step transfer scheme - see OpenZeppelin Ownable2Step contract - is recommended to provide an extra level of control for unintended ownership transfers.

Status

Addressed in commits `4384b578135adf15dd924a13a68946d1d371c9d0` and `06c857f8303fca5ce835d604b14867dc26ed9143`.

A03: `ZivoeGTC::updateTimelock` function is unprotected

[Severity: High | Difficulty: Medium | Category: Input Validation]

A call to `ZivoeGTC::updateTimelock()` directly updates the `_timelock` variable to the provided address, which grants a great extent of privileges to the transferred address. Similarly to A03, a significant amount of functionality is unusable in case of transfer to `address(0)` or to an invalid address.

Scenario

1. A governance proposal to update the `_timelock` to an invalid address (an unnoticed typing error, for instance) gets accepted and executed.
2. It is no longer possible to make new proposals or execute proposals, which leads to the inoperability of the protocol,

Recommendation

A 2-step transfer scheme - see OpenZeppelin Ownable2Step contract - is recommended to provide an extra level of control for unintended ownership transfers.

Status

Addressing A04 led to the deletion of this function.

A04: `ZivoeGTC::updateTimelock` function leads to loss of functionality in `ZivoeTranches`

[Severity: Medium | Difficulty: Medium | Category: Usability]

A call to `ZivoeGTC::updateTimelock()` updates the `_timelock` variable to the provided address in the `ZivoeGTC` contract. However, in the `ZivoeGlobals` contract it is not possible to update the `TLC` variable, which initially - according to the deployment scheme - stores the same address as `_timelock`. Some functions in `ZivoeTranches`, responsible for modifying system parameters, have the modifier `onlyGovernance`, which checks that the `msg.sender` is the `TLC` address (in `ZivoeGlobals`). Therefore, if the `_timelock` in `ZivoeGTC` is updated, since it is impossible to update the `TLC` in `ZivoeGlobals`, all the functionality restricted to `onlyGovernance` in `ZivoeTranches` is no longer available.

Recommendation

Either:

1. Remove the `updateTimelock` function in `ZivoeGTC`, making it impossible to update the `_timelock`.
2. Add an `updateTLC(address tlc)` function in `ZivoeGlobals`, guaranteeing that the call to this function is done only when updating the `_timelock` to a new `address` and that the new `TLC` must be the same as the new `_timelock`.

Status

Addressed with recommendation 1. in commit a3c7f5bd08a6264acfd3166fd986a0fd4300027.

A05: `ZivoeYDL::updateRecipients()` does not check for `address(0)`

[Severity: Medium | Difficulty: Medium | Category: Input Validation]

`ZivoeYDL::updateRecipients()` function, responsible for updating the recipients of protocol yield, does not check if some of the recipient addresses are `address(0)`.

Scenario

1. A governance proposal to update the recipients that contain the `address(0)` in the list of recipients gets accepted and executed.
2. If the `distributedAsset`:
    i. Supports transfers to `address(0)`, the yield is lost.
    ii. Reverts on transfers to `address(0)`, the yield distribution is blocked.

Recommendation

Add a sanity check for `address(0)` in the function `ZivoeYDL::updateRecipients()`.

Status

Addressed in commit [d3b2367b339cca79680678d76342b9ddd234e3fd]().

A06: Far future ending of reward vesting schedule can be constructed

[Severity: Low | Difficulty: High | Category: Input Validation]

`ZivoeRewardsVesting::createVestingSchedule` accepts time values for `daysToCliff` and `daysToVest` that may be a very distant time in the future. These values can be big enough to cause `vestingPerSecond` to be `0`, and, consequently, `amounWithdrawable` to be also `0` if the vesting schedule is not finished yet.

```
vestingScheduleOf[account].vestingPerSecond = amountToVest /
                                                (daysToVest * 1 days);
```

In fact, `vestingPerSecond` and `amountWithdrawable` can be `0` even with a reasonable `daysToVest` value, in case the `amountToVest` is small enough. However, if such a scenario occurs, the account can still withdraw the assets at the end of the vesting schedule. Whereas, in case when `daysToVest` is considerably large, the account will never be able to withdraw its tokens.

```
function amountWithdrawable(address account) public view returns (uint256 amount)
{
        if (block.timestamp < vestingScheduleOf[account].cliff) { return 0; }
        if (
            block.timestamp >= vestingScheduleOf[account].cliff &&
            block.timestamp < vestingScheduleOf[account].end
        ) {
            return vestingScheduleOf[account].vestingPerSecond * (
                block.timestamp - vestingScheduleOf[account].start
            ) - vestingScheduleOf[account].totalWithdrawn;
        }
        else if (block.timestamp >= vestingScheduleOf[account].end) {
            return vestingScheduleOf[account].totalVesting -
                    vestingScheduleOf[account].totalWithdrawn;
        }
        else { return 0; }
}
```

Recommendation

Designating and checking for a maximum time ahead can mitigate this issue.

Status

Addressed in commit `802ccb08f564177cbbec82b16f11565ea0635276`.

A07: `ZivoeTLC::minDelay` can be unreasonably large

[Severity: High | Difficulty: Medium | Category: Input Validation]

The `minDelay` variable in `ZivoeTLC` represents the amount of time a valid proposal needs to wait until it can be executed. However, neither the constructor nor the function `updateDelay` check that the value being assigned to this variable is a reasonable amount of time.

Scenario

1. A governance proposal to update the `minDelay` to an unreasonably large amount of time gets accepted and executed.
2. The variable `minDelay` gets assigned to that value.
3. All the new valid proposals submitted to the `timelock` will have to wait that extremely amount of time until they are executed.
4. The protocol gets locked.

Recommendation

Add an upper bound check on the argument every time the variable `minDelay` gets assigned to that argument.

Status

The check in the constructor was addressed in commit `0da3ecde661cd2e98f00896498d7507f4647a543`, and the check in the function `updateDelay` was addressed in commit `a885ac5134447092aaadc585db3ec4c90aa22a07`.

A08: Governance can `revokeRole` of `PROPOSER` or `EXECUTOR` locking the protocol

[Severity: High | Difficulty: Medium | Category: Usability/Protocol Invariants]

`ZivoeTLC` inherits from `AccessControl` contract from OpenZeppelin library and has the following roles: `TIMELOCK_ADMIN_ROLE`, `PROPOSER_ROLE`, `EXECUTOR_ROLE` and `CANCELLER_ROLE`.

According to the deployment scheme, the governance contract will have the `PROPOSER_ROLE`, whereas `address(0)` will have the `EXECUTOR_ROLE` for public execution of proposals. However, in the `ZivoeTLC` constructor, the `ZivoeTLC` contract is granted the `TIMELOCK_ADMIN_ROLE`. Additionally, the `TIMELOCK_ADMIN_ROLE` is the `admin` of all the other roles, meaning it can grant or revoke roles for any address.

Scenario

1. Governance is the only address with `PROPOSER_ROLE` (or `address(0)` is the only address with `EXECUTOR_ROLE`).
2. A governance proposal to `revokeRole()` of `PROPOSER_ROLE` to `ZivoeGovernor` contract (or `EXECUTOR_ROLE` to `address(0)`) goes through and gets executed.
3. It is no longer possible to make (respectively execute) proposals to the timelock.
4. The protocol gets locked.

Recommendation

There are two possible solutions:

1. Remove the `TIMELOCK_ADMIN_ROLE` from the `ZivoeTLC` contract. This solution implies that it is impossible to grant roles to any address in the future.
2. Guarantee when revoking the `PROPOSER_ROLE` or the `EXECUTOR_ROLE` that another address has that role. Since they are crucial roles for the protocol, it is essential that some other address can propose or execute proposals.

Status

Addressed via solution 1. in commit `f646f30e85af28ba57eaea46ef46d00c6fe3c61b`.

A09: Governance can update governance settings to unreasonable values griefing the protocol

[Severity: High | Difficulty: Medium | Category: Input Validation]

Some governance settings, such as `_votingDelay`, `_votingPeriod`, `_proposalThreshold` and `_quorumNumerator`, can be updated through governance proposals. The setters for these variables do not have any upper bound, meaning that a governance proposal to update any of these variables to an unreasonable value can go through, making the protocol inoperable since the proposing or execution of proposals gets compromised.

Recommendation

Override the setting of these variables and enforce upper bounds on the arguments

Status

The functions `setVotingDelay`, `setVotingPeriod` and `setProposalThreshold` were properly overridden and capped in commit `cac51028315455bb737b17763897c1ac78717e5a`. The `updateQuorumNumerator` capped by 30 in commit `5f10e68966507caeec4abcefeefb151b1c00125e`.

A010: Upper bound updates may go below lower bounds

[Severity: Low | Difficulty: Medium | Category: Invariants violation]

Both functions `updateMaxZVEPerJTTMint()` and `updateUpperRatioIncentiveBIPS()` in `ZivoeTranches` contract do not check if upper limits are higher than lower limits, making it possible to update them incorrectly.

Status

Addressed in commit [`7f03d027fc149f54d069ea93c1cdae56e82cb6ff`](7f03d027fc149f54d069ea93c1cdae56e82cb6ff).

Informative Findings

B01: Optimization in `Rewards` and `RewardsVesting`

[Severity: - | Difficulty: - | Category: Gas Optimization]

Both functions `getRewards` and `getRewardsAt` have the modifier `updateReward(msg.sender)`.

```
function getRewards() public updateReward(_msgSender()) {
     for (uint256 i = 0; i < rewardTokens.length; i++) { getRewardAt(i); }
 }
```

```
function getRewardAt(uint256 index) public nonReentrant updateReward(_msgSender())
```

Besides, the function `getRewards` calls the function `getRewardsAt(i)` for each token `i` in `rewardTokens`, and the `updateReward` modifier updates all token rewards for every token in `rewardTokens`.

Scenario

If the contract has 10 `rewardTokens` (the maximum), the modifier `updateReward` is called 11 times. This would lead to 110 iterations to update the `tokens`, although the meaningful updates would only happen in the first call of `updateReward` (the first 10 iterations).

Recommendation

One of the two following solutions:

1. Change the visibility of `getRewardAt(uint256 index)` to `internal` and remove the modifier `updateReward(_msgSender())` in this function.

2. Keep the visibility of `getRewardAt(uint256 index) public`. Implement a function/modifier that only updates the token rewards of a specific token - `updateRewardAt(address account, uint256 index)`. Replace the modifier `updateReward` in the function `getRewardAt` with `updateRewardAt` and remove the modifier `updateReward` from the function `getRewards`.

Status

Addressed following approach 1. in commit [f7d8223388ce331374d499cd1e6f56b4e5f7ff02](f7d8223388ce331374d499cd1e6f56b4e5f7ff02).

B02: Pre-compute and reuse value in `ZivoeYDL::distributeYield()`

[Severity: - | Difficulty: - | Category: Gas Optimization]

Values `_protocol[i] * splitBIPS / BIPS` and `_residual[i] * splitBIPS / BIPS` are used repeatedly in the function.

Recommendation

These values can be calculated once as in

```
uint256 splitValue = _protocol[i] * splitBIPS / BIPS
```

and reused in the function. Moreover, this precomputed value can be further reused to replace `_protocol[i] * (BIPS - splitBIPS) / BIPS` with `_protocol[i] - splitValue` and similarly for the residuals calculation.

Status

Addressed in commit c05bba20cc2ccdb69e68c42b53a4a04b8cc8badb.

B03: `ZivoeYDL::distributeYield()` may revert and lock yield distribution for black listed addresses

[Severity: - | Difficulty: - | Category: Usability]

`ZivoeYDL::distributeYield()` function contains two loops that transfers `distributedAsset` `ERC20` tokens to a `recipient` that can be set via `updateRecipients` function. However, `distributedAsset` tokens, which are primarily designated to be a stablecoin, may be blocking transfers to the `recipient` and hence revert all the yield distribution operations done by the loop previously blocking all the yield distribution operations.

This situation may be temporarily mitigated by removing the black listed `recipient` and updating the `recipient` list or switching to another stablecoin where the recipient is not blacklisted. However those mitigations don't prevent future blacklisted recipients from blocking the yield distribution.

B04: A reward token cannot be removed from the list of reward tokens once pushed

In `ZivoeRewards` and `ZivoeRewardsVesting` contracts, the function `addReward()` adds a new reward token to the list of rewards being distributed to stakers. However, there is no means of removing a reward token from the list of reward tokens.

Recommendation

Possible effects of removing a token from the list should be investigated and implemented if plausible.

An approach to remove a token may be to transfer the contents of `rewardTokens` to a temporary array, reset `rewardTokens` and transfer back from the temp array but skip the token to be removed. Also, update the `rewardsDuration` of the token to 0.

An additional idea is to ensure that it is only possible to remove a reward token after its period is finished.

B05: `ZivoeYDL::distributeYield()` can run out of gas for a sufficiently large list of recipients

[Severity: - | Difficulty: - | Category: Gas Cost]

Two different lists of recipients are iterated over in the `ZivoeYDL::distributeYield()` function, which may be exploited to revert via gas outage if the list of recipients is updated with a necessarily large number of recipients.

Recommendation

A gas usage investigation must be done to determine how long the list of recipients and residuals can be without the function `distributeYield` running out of gas. If lists of recipients greater than that are plausible, then the mechanism of distributing yield must be modified. One possible solution is to implement a data structure that stores the state of yield distribution, allowing to distribute the yield in slices of the recipients.