Security Audit Report

Zivoe Vault Ethereum

Delivered: March 31, 2025

Table of Contents

Disclaimer

This report does not constitute legal or investment advice. You understand and agree that this report relates to new and emerging technologies and that there are significant risks inherent in using such technologies that cannot be completely protected against. While this report has been prepared based on data and information that has been provided by you or is otherwise publicly available, there are likely additional unknown risks which otherwise exist. This report is also not comprehensive in scope, excluding a number of components critical to the correct operation of this system. This report is for informational purposes only and is provided on an "as-is" basis and you acknowledge and agree that you are making use of this report and the information contained herein at your own risk. The preparers of this report make no representations or warranties of any kind, either express or implied, regarding the information in or the use of this report and shall not be liable to you or any third parties for any acts or omissions undertaken by you or any third parties based on the information contained herein.

Smart contracts are still a nascent software arena, and their deployment and public offering carries substantial risk.

Finally, 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.

Executive Summary

Zivoe engaged Runtime Verification Inc. to perform a security audit of its Zivoe Vault functionality, focusing on the ZivoeVault, ZivoeRouter, and OCT_Convert contracts. The goal of the audit was to assess the security and correctness of the implementation, uncover any exploitable vulnerabilities, and provide recommendations to strengthen the reliability of the system.

Zivoe is building a decentralized credit protocol, and the Zivoe Vault plays a central role in the system's asset management lifecycle. It facilitates the interaction between user-deposited capital and underlying investment strategies, providing routing logic and accounting mechanisms to enable composable, rules-based treasury deployment.

The audit was conducted over one calendar week (March 24-31, 2025). Runtime Verification performed a design review to assess the high-level intent and security-critical invariants of the vault system, followed by a focused manual review of the Solidity implementation. This process was supported by Kontrol, our formal verification tool, which we used to write and check invariants across symbolic state transitions.

All findings have been clearly documented in the following sections, and suggestions have been provided to improve correctness, maintainability, and security of the Zivoe Vault system. Any responses or fixes submitted by the client will be reviewed and noted as part of the report's finalization.

Scope

The scope of the audit is limited to the code contained in two GitHub repositories provided by the client, one of which is public and the other currently private. Within these repositories, specific files and contracts were highlighted as in-scope for this engagement. The repositories, contracts, and commit information are detailed below:

Zivoe Vault Repository (private)
https://github.com/Zivoe/zivoe-vault
Commit: 8efa4c9db0c9c53e1892434f536d0f08e288742a

src/ZivoeVault.sol: Core contract handling asset deposits, withdrawals, and interactions with strategies.

src/ZivoeRouter.sol: Contract managing routing logic between user interactions and the vault.

Zivoe Core Foundry Repository (public)
https://github.com/Zivoe/zivoe-core-foundry/commit/7acab6ae1b04e6acde7b20fefdac6c190037b102

src/lockers/OCT/OCT_Convert.sol: Locker contract facilitating conversion logic related to the OCT strategy.

The comments provided in the code, a general description of the project, unit tests within the repositories, and additional documentation provided by the client were used as reference material.

The audit is limited in scope to the artifacts listed above. Off-chain components, auto-generated code, deployment and upgrade scripts, or any client-side logic are excluded from the scope of this engagement.

Commits addressing the findings presented in this report have also been analyzed to ensure that the relevant issues have been properly resolved.

Methodology

Although manual code review cannot guarantee the discovery of all possible security vulnerabilities, as noted in our Disclaimer, we followed a systematic approach to make this audit as thorough and impactful as possible within the allotted time frame.

The audit engagement spanned one calendar week and began with a focused design review. We allocated the first day to analyzing the architecture and intended functionality of the Zivoe Vault system. This included reasoning about the interactions between the vault, router, and locker contracts, and identifying key security-critical properties that should be upheld throughout the system's lifecycle.

Following the design review, we conducted a thorough manual code review of the in-scope contracts: ZivoeVault, ZivoeRouter, and OCT_Convert. This process was aided by our internal formal verification tool, Kontrol, which enables symbolic execution of Solidity code. Where appropriate, we defined formal invariants and used Kontrol to verify them under a wide range of symbolic inputs and system states. This helped us reason exhaustively about key properties and surface potential edge cases.

Findings presented in this report stem from a combination of:

- Manual inspection of the Solidity source code.
- Design-level reasoning about contract interactions and system invariants.
- Symbolic proofs and property-based assertions executed using Kontrol.

In addition to identifying bugs and vulnerabilities, we also evaluated gas usage patterns, reviewed edge-case handling, and provided recommendations for code clarity and safety improvements.

Throughout the engagement, we held internal discussions among auditors to cross-review the findings and validate risk assessments.

Internal Contract Summaries

The following are summaries of the behavior of the contracts within scope. We describe below what each function of the contract does and provide a mathematical model capturing the storage updates after the function successfully returns. In this model, we use `S` to denote the state before the function execution and `S'` to denote state after the function execution.

Note that for functions that interact with the external `ZivoeRewards` contract, the model abstracts away updates to the rewards and checkpoints. It also assumes that the staking token is zSTT (the senior `ZivoeTranchesToken`), which is consistent with the version of `ZivoeRewards` deployed on chain. The notation `standardize(amount, stablecoin)` used below refers to the function of the same name from the `ZivoeGlobals` contract, which standardizes `amount` to 18 decimals, assuming that it's represented in the precision of `stablecoin`.

ZivoeVault

An ERC-4626 vault that stakes zSTT (the senior `ZivoeTrancheToken`) into stSTT (the `ZivoeRewards` contract).

Invariants

- `this.totalSupply()` is the sum of `this.balanceOf(user)` across all users of the protocol.
  - As a corollary, for a balance of a user to increase, either the total supply must increase, or the balance of a different user (or users) must decrease by the same amount. Likewise, for the balance of a user to decrease, either the total supply must decrease, or the balance of a different user (or users) must increase by the same amount.

- `zSTT.balanceOf(this) = 0` unless an address independently transfers zSTT to this contract (every function either keeps the balance the same or sets it to 0).

`totalAssets()`

Returns the total number of assets deposited into the vault, given by the vault's stSTT balance.

Note that the default implementation of this function in OpenZeppelin's `ERC4626` contract instead uses the vault's balance in the underlying asset. However, since the vault passes all deposited zSTT to stSTT and doesn't itself maintain a zSTT balance (unless zSTT is independently transferred to it), the function is overloaded to use the stSTT balance instead.

`_deposit(caller, receiver, assets, shares)`

Deposits zSTT into the vault, minting shares to the receiver. This internal function is called by the `deposit` and `mint` functions inherited from OpenZeppelin's `ERC4626` contract:

- `deposit` receives `assets` as argument and calculates `shares = assets * (totalSupply() + 1) / (totalAssets() + 1)` (rounded down)
- `mint` receives `shares` as argument and calculates `assets = shares * (totalAssets() + 1) / (totalSupply() + 1)` (rounded up)

Note that the implementation is similar to the default implementation of `_deposit` in `ERC4626`, with the additional step of staking the zSTT into `ZivoeRewards`.

Summary

- Transfer `assets` zSTT from `caller` to vault.
- Mint the `shares` to `receiver`.
- Stake `assets` in stSTT (see `ZivoeRewards.stake`).

Model

```
S'.zSTT.balanceOf(caller) := S.zSTT.balanceOf(caller) - assets
S'.zSTT.balanceOf(stSTT) := S.zSTT.balanceOf(stSTT) + assets
S'.this.totalSupply() := S.this.totalSupply() + shares
S'.this.balanceOf(receiver) := S.this.balanceOf(receiver) + shares
S'.stSTT.totalSupply() := S.stSTT.totalSupply() + assets
S'.stSTT.balanceOf(this) : = S.stSTT.balanceOf(this) + assets
```

`_withdraw(caller, receiver, owner, assets, shares)`

Withdraws zSTT from the vault, burning shares from the receiver. This internal function is called by the `withdraw` and `redeem` functions inherited from OpenZeppelin's `ERC4626` contract:

- `withdraw` receives `assets` as argument and calculates `shares = assets * (totalSupply() + 1) / (totalAssets() + 1)` (rounded up)
- `redeem` receives `shares` as argument and calculates `assets = shares * (totalAssets() + 1) / (totalSupply() + 1)` (rounded down)

Note that the implementation is similar to the default implementation of `_withdraw` in `ERC4626`, with the additional step of withdrawing the zSTT from `ZivoeRewards`.

Summary

- Burn `shares` from `owner`.
- Withdraw `assets` from stSTT (see `ZivoeRewards.withdraw`).
- Transfer `assets` zSTT from vault to `receiver`.

Model

```
S'.this.totalSupply() := S.this.totalSupply() - shares
S'.this.balanceOf(owner) := S.this.balanceOf(owner) - shares
S'.stSTT.totalSupply() := S.stSTT.totalSupply() - assets
S'.stSTT.balanceOf(this) := S.stSTT.balanceOf(this) - assets
S'.zSTT.balanceOf(stSTT) := S.zSTT.balanceOf(stSTT) - assets
S'.zSTT.balanceOf(receiver) := S.zSTT.balanceOf(receiver) + assets
```

`compound(stablecoin)`

Claims rewards from stSTT and stakes `stablecoin` back as zSTT. Note that all rewards are claimed, but only `stablecoin` is re-staked. This means that if there are multiple reward tokens, `compound` must be called multiple times to re-stake all of them. The function can also be called with a stablecoin that is not an actual reward token of stSTT. In that case, it will still claim all reward tokens but not re-stake any, unless the vault already holds some of the given stablecoin. See also issue A01.

Summary

- Get rewards for the vault from stSTT (see `ZivoeRewards.getRewards`).
- Deposit the vault's entire balance of `stablecoin` into zSTT (see `ZivoeTranches.depositSenior`).
- Stake the vault's entire zSTT balance into stSTT (see `ZivoeRewards.stake`).

Model

For each `rewardsToken` in `rewardTokens`, where `rewards` is the amount of rewards accumulated by the vault in `rewardsToken`:

```
S'.rewardsToken.balanceOf(stSTT) := S.rewardsToken.balanceOf(stSTT) - rewards
```

For `rewardsToken != stablecoin`:

```
S'.rewardsToken.balanceOf(this) := S.rewardsToken.balanceOf(this) + rewards
```

Where `rewards` is the rewards amount for `stablecoin`:

```
S’.stablecoin.balanceOf(this) := 0
S’.stablecoin.balanceOf(DAO) := S.stablecoin.balanceOf(this) + rewards

S’.zSTT.totalSupply() :=
    S.zSTT.totalSupply() +
    standardize(S.stablecoin.balanceOf(this) + rewards, stablecoin)
S’.zSTT.balanceOf(this) := 0
S’.zSTT.balanceOf(stSTT) :=
    S.zSTT.balanceOf(stSTT) +
    S.zSTT.balanceOf(this) +
    standardize(S.stablecoin.balanceOf(this) + rewards, stablecoin)

S’.stSTT.totalSupply() :=
    S.stSTT.totalSupply() +
    S.zSTT.balanceOf(this) +
    standardize(S.stablecoin.balanceOf(this) + rewards, stablecoin)
S’.stSTT.balanceOf(this) :=
    S.stSTT.balanceOf(this) +
    S.zSTT.balanceOf(this) +
    standardize(S.stablecoin.balanceOf(this) + rewards, stablecoin)
```

Where `incentives` is the ZVE rewards for depositing in the senior tranche:

```
S’.ZVE.balanceOf(ZVT) := S.ZVE.balanceOf(ZVT) - incentives
S’.ZVE.balanceOf(this) := S.ZVE.balanceOf(this) + incentives
```

`passThrough(asset)`

Transfers the vault's entire `asset` balance to the DAO, as long as `asset != zSTT`.

Model

```
S’.asset.balanceOf(this) := 0
S’.asset.balanceOf(DAO) := S.asset.balanceOf(DAO) + S.asset.balanceOf(this)
```

ZivoeRouter

Router that automates the typical workflow for users to interact with the `ZivoeVault`.

Invariants

- This contract does not maintain any balance in zSTT, stSTT or the `ZivoeVault`, nor in any of the stablecoins it operates with, unless another account independently transfers tokens or shares to it.

`depositVault(stablecoin, amount)`

Receives stablecoins from the user and deposits them into the vault as zSTT.

Note that only `amount` of `stablecoin` is transferred from the caller and converted into zSTT, but the router deposits its entire zSTT balance into the vault. This means that if the router already has a zSTT balance for whatever reason, that will also be deposited in the user's name. This shouldn't happen during normal operation of the contract, only if another account independently transfers zSTT to the router (probably by accident). See also issue A03.

Summary

- Transfer `amount` of `stablecoin` from `msg.sender` to the router.
- Deposit `amount` of `stablecoin` into zSTT (see `ZivoeTranches.depositSenior`).
- Transfer the router's entire balance of ZVE to `msg.sender`.
- Deposit the router's entire balance of zSTT to vault in the name of `msg.sender` (see `ZivoeVault.deposit`).

Model

```
S’.stablecoin.balanceOf(msg.sender) := S.stablecoin.balanceOf(msg.sender) - amount
S’.stablecoin.balanceOf(this) := S.stablecoin.balanceOf(this)
S’.stablecoin.balanceOf(DAO) := S.asset.balanceOf(DAO) + amount

S’.zSTT.totalSupply() := S.zSTT.totalSupply() + standardize(amount, stablecoin)
S’.zSTT.balanceOf(this) := 0
S’.zSTT.balanceOf(stSTT) :=
    S.zSTT.balanceOf(stSTT) +
    S.zSTT.balanceOf(this) +
    standardize(amount, stablecoin)

S’.stSTT.totalSupply() :=
    S.stSTT.totalSupply() +
    S.zSTT.balanceOf(this) +
    standardize(amount, stablecoin)
S'.stSTT.balanceOf(VLT) : =
    S.stSTT.balanceOf(VLT) +
    S.zSTT.balanceOf(this) +
    standardize(amount, stablecoin)
```

Where `incentives` is the ZVE rewards for depositing in the senior tranche:

```
S'.ZVE.balanceOf(ZVT) := S.ZVE.balanceOf(ZVT) - incentives
S'.ZVE.balanceOf(msg.sender) := S.ZVE.balanceOf(msg.sender) + incentives
```

For `shares = (S.zSTT.balanceOf(this) + standardize(amount, stablecoin)) * (S.VLT.totalSupply() + 1) / (S.stSTT.balanceOf(VLT) + 1)`:

```
S'.VLT.totalSupply() := S.VLT.totalSupply() + shares
S'.VLT.balanceOf(msg.sender) := S.VLT.balanceOf(msg.sender) + shares
```

`depositWithPermit(stablecoin, amount, deadline, v, r, s`

Same as above, but using `permit` instead of `approve` for the initial transfer from `msg.sender`.

OCT_Convert

This contract is meant to support the deprecation of the junior tranche by converting zJTT (the junior `ZivoeTrancheToken`) to zSTT (the senior `ZivoeTrancheToken`), using underlying stablecoins transferred from the `ZivoeDAO` in advance. It also includes functionality to withdraw zSTT into the underlying stablecoin, to be used to withdraw leftover liquidity after the process is concluded.

Invariants

- `zSTT.balanceOf(this) = 0` unless an address (including the DAO) independently transfers zSTT to this contract (neither `convertTranche` nor `withdrawTranche` leave zSTT in the contract). The same applies to zJTT.
- `convertTranche` and `withdrawTranche` always decrease the stablecoin balance of the contract, never increase.

`updateWhitelist(user, status)`

Updates depositor status of `user` to `status`.

`convertTranche(amount, stablecoin)`

Converts zJTT into zSTT. This depends on the `OCT_Convert` contract having a preexisting `stablecoin` balance, so the necessary funds must have been transferred from the `ZivoeDAO` contract before the function is called. The caller must also have been registered as a depositor in advance.

Summary

- Transfer `standardize(amount, stablecoin)` of zJTT from `msg.sender` to this locker.
- Burn `standardize(amount, stablecoin)` zJTT.
- Deposit `amount` of `stablecoin` into zSTT (see `ZivoeTranches.depositSenior`).
- Transfer `standardize(amount, stablecoin)` of zSTT to `msg.sender`.

Model

```
S’.zJTT.balanceOf(msg.sender) := S.zJTT.balanceOf(msg.sender) - standardize(amount,
stablecoin)
S’.zJTT.totalSupply() := S.zJTT.totalSupply() - standardize(amount, stablecoin)
```

```
S'.stablecoin.balanceOf(this) := S.stablecoin.balanceOf(this) - amount
S'.stablecoin.balanceOf(DAO) := S.stablecoin.balanceOf(DAO) + amount
S'.zSTT.totalSupply() := S.zSTT.totalSupply() + standardize(amount, stablecoin)
S’.zSTT.balanceOf(msg.sender) := S.zSTT.balanceOf(msg.sender) + standardize(amount,
stablecoin)
```

Where `incentives` is the ZVE rewards for depositing in the senior tranche:

```
S'.ZVE.balanceOf(ZVT) := S.ZVE.balanceOf(ZVT) - incentives
S'.ZVE.balanceOf(this) := S.ZVE.balanceOf(this) + incentives
```

`withdrawTranche(amount, stablecoin)`

Withdraws `amount` of `stablecoin` from the contract by burning the equivalent amount of zSTT.

Summary

- Transfer `standardize(amount, stablecoin)` zSTT from caller to this locker.
- Burn `standardize(amount, stablecoin)` of zSTT.
- Transfer `amount` of `stablecoin` to `msg.sender`.

Model

```
S’.zSTT.balanceOf(msg.sender) := S.zSTT.balanceOf(msg.sender) - standardize(amount,
stablecoin)
S’.zSTT.totalSupply() := S.zSTT.totalSupply() - standardize(amount, stablecoin)
S’.stablecoin.balanceOf(this) := S.stablecoin.balanceOf(this) - amount
S’.stablecoin.balanceOf(msg.sender) := S.stablecoin.balanceOf(msg.sender) + amount
```

Kontrol Formal Verification

In addition to the code review, we have also adapted some of the tests in the repository to be symbolically executed with Runtime Verification's formal verification tool Kontrol. Kontrol is designed to integrate seamlessly with Solidity-based projects, enabling developers to write property-based tests in Solidity and leverage symbolic execution to verify them, thus ensuring that smart contracts behave as intended under all possible inputs and scenarios.

Symbolic Testing with Kontrol

Kontrol's functionality is integrated with the Forge testing framework, a component of the Foundry toolkit for smart contract development. Kontrol tests are written in the same way as Forge parameterized tests, but instead of using fuzzing to run each test on randomly-generated inputs, Kontrol executes the tests symbolically. This means that test parameters are interpreted as symbolic variables with no concrete value associated with them. Kontrol then uses symbolic execution to step through the test, exploring all possible execution paths, employing logical and mathematical reasoning to determine if the assertions are satisfied on every path. We can also use specialized cheatcodes to make the storage of the relevant contracts symbolic, enabling tests to start from an arbitrary initial state.. This gives stronger guarantees than fork testing, for example, since rather than executing the test starting from one specific initial state, the results hold for any state that the protocol can be in. In this way, if the test passes, we know that it passes for any test input and starting state.

(Note that in practice, a number of assumptions might need to be made, either to eliminate invalid initial states that are impossible or where the property that we are testing for doesn't apply, or to avoid corner cases that complicate symbolic execution. A common example is to bound the value of test inputs and storage variables to avoid reverts due to overflows. In these tests, for example, we assume that token amounts and balances will be under `2 ** 96`.)

`ZivoeVault` Tests

We have submitted a pull request with the Kontrol version of the `ZivoeVault` tests at https://github.com/Zivoe/zivoe-vault/pull/10. The main changes to the original version of the tests are the following:

- Initialization functions for the Zivoe core contracts making their storage symbolic. This includes initializing the balance of the contracts in the relevant tokens to symbolic values. This initialization is implemented in `Utility_Kontrol`, and is called via `_deployContracts` as part of the test contracts' `setUp` function.
- `test_Vault_deposit` and `test_Vault_withdraw` in `Test_Vault_Kontrol` modified to take the deposit/withdraw amount and shares receiver as symbolic test parameters, rather than the fixed concrete values in the original tests.
- Assertions added to `test_Vault_deposit` and `test_Vault_withdraw` checking that the state changes after the functions are called match the mathematical model presented in this report. The tests save snapshots of the state before and after the function is called (corresponding to S and S') and then compare these values to ensure that the expected state updates occur in all cases.

As mentioned above, assumptions were also added to the initialization and tests to eliminate invalid initial states and inputs or bound values to reasonable sizes. Some simplifying assumptions are also made to reduce the scope of the formal verification; for example, since `test_Vault_deposit` and `test_Vault_withdraw` are not concerned with reward updates in the `ZivoeRewards` contract, we run the tests with no reward tokens for simplicity.

The tests `test_Vault_init` and `test_Vault_deposit` have been confirmed to pass in Kontrol. In its current state, `test_Vault_withdraw` still needs to be refined further in order to pass, requiring additional assumptions. The other tests, such as `test_Vault_compound` and `test_Vault_passThrough`, can be similarly modified using the other tests as models, in order to verify in the same way that they perform the correct state updates.

Reproducing the Proofs

To reproduce the results of this verification locally, follow the steps below:

1. Install Kontrol

```
bash <(curl https://kframework.org/install)
kup install kontrol
```

2. Run the proofs

```
export FOUNDRY_PROFILE=kprove
kontrol build
kontrol prove
```

Foundry profile `kprove` is configured in the `foundry.toml` file to build the project for use with Kontrol. The `kontrol.toml` file contains a set of options and flags that Kontrol will execute with. The file can be edited to change these options, or turn flags on or off. Refer to the Kontrol documentation if you want to learn more about Kontrol options.

Findings

This section contains all issues identified during the audit that could lead to unintended behavior, security vulnerabilities, or failure to enforce the protocol’s intended logic. Each issue is documented with a description, potential impact, and recommended remediation steps.

A01: `compound` allows withdrawing rewards without re-staking them, opening up a griefing attack

Severity: Medium | Difficulty: Low | Recommended Action: Fix Code | Not addressed by client

When `ZivoeVault.compound(stablecoin)` is called, all rewards for `ZivoeVault` are withdrawn, but only the `stablecoin` rewards are re-staked in the `ZivoeRewards` contract. Any other reward tokens remain in `ZivoeVault`, until one of two things happen:

- `compound` is called again with this reward token instead, staking it back into `ZivoeRewards` (as long as it's a stablecoin whitelisted in `ZivoeGlobals`, otherwise `ZivoeTranches.depositSenior` reverts).
- `passThrough` is called with this reward token, sending it to the `ZivoeDAO` (as long as it's not zSTT).

The main issue is that this behavior can be used for a griefing attack where an attacker calls `compound` to withdraw a reward from `ZivoeRewards` to `ZivoeVault` without re-staking it, then calls `passThrough` to send these tokens directly to the `ZivoeDAO`, thus preventing the compounding of the rewards.

Scenario

1. Attacker calls `compound(stablecoin)` for `stablecoin != address(USDC)` (note that `stablecoin` can be any token, not necessarily one of the reward tokens registered in `ZivoeRewards`).
2. USDC rewards are transferred from `ZivoeRewards` to `ZivoeVault` without being re-staked.
3. Attacker calls `passThrough(USDC)`, sending all USDC rewards to the `ZivoeDAO`.

Recommendation

Ideally, `compound` would be able to either (a) withdraw only the reward token that it will re-stake, or (b) iterate over all reward tokens withdrawn and re-stake all of them. However, neither functionality is supported by `ZivoeRewards`. Therefore, the best option is to prevent an attacker from sending the stablecoin to the DAO via `passThrough`. One way to do that would be to limit the function to only be called by a designated set of keepers that can be trusted to not call it with a stablecoin that could be compounded. A better option might be to require in `passThrough` that the asset is not one of the stablecoins whitelisted in `ZivoeGlobals`. Since `depositSenior` can only mint zSTT from whitelisted stablecoins, this would block those assets and only those assets that can instead be re-staked by `compound`.

A02: `IERC20Permit.permit` can be front-run to prevent `depositWithPermit` from completing

Severity: Low Difficulty: Low Recommended Action: Document Prominently

Not addressed by client

`ZivoeRouter.depositWithPermit` uses the `IERC20.permit` function to approve the transfer of stablecoins from the user to the router. However, an attacker can front-run the transaction and call `permit` with the same arguments. This correctly sets the allowance for the router, however when `depositWithPermit` is called by the user, the call to `permit` will revert because the nonce has been incremented.

Although there is no obvious way for an attacker to benefit from it, this could be used as a griefing attack causing the user to waste gas without having their deposit go through. However, since the allowance is set regardless, the user could then call the regular `depositVault` function instead to still perform the deposit.

If the suggestion in issue B03 is implemented, however, a caller could call `depositWithPermit` on behalf of a signer that provided their signature. In that case, if the transaction was prevented by an attacker in this way, the caller would not be able to circumvent the issue by calling `depositVault`, since that function can only be used to deposit tokens for the `msg.sender`. In that case, the signer would have to either call `depositVault` themselves or provide a new signature for the updated nonce (although the latter could be front-run again).

Recommendation

Although the `permit` call could be wrapped in a `try` - `catch` block to allow `depositWithPermit` to continue execution as long as the router has an allowance (see this observation in OpenZeppelin's `IERC20Permit`), this would be risky in the case B03 is implemented. In the case a user has given the router an allowance, anyone could call `depositWithPermit`, even with an invalid signature, to deposit tokens from that user up to the allowance against their will.

In the case a user is depositing for themselves, they should be aware that this is a possibility, albeit unlikely, and be prepared to fall back to `depositVault` if it happens. If issue B03 is implemented, any application that makes use of `depositWithPermit` to deposit on behalf of another account should consider this risk and provide an alternative method for users to complete the deposit.

A03: Stablecoins sent to the `ZivoeRouter` cannot be recovered

Severity: Low | Recommended Action: Fix Code | Not addressed by client

`ZivoeRouter.depositVault` receives some amount of a stablecoin and converts it into zSTT, then deposits its entire zSTT balance into the vault on behalf of the caller. This means that if for some reason the router has a zSTT balance (for example, if someone has transferred tokens to it by mistake), this extra balance will go to the next user who calls `depositVault`.

However, the same is not true for the stablecoin, since the function will only convert the provided amount of stablecoin into zSTT. Therefore, if the router happens to have a stablecoin balance, there is no way to get these out.

Recommendation

Rather than calling `depositSenior` with only the provided amount of the stablecoin, `depositVault` can call it with its entire current balance. This will allow any stablecoins accidentally transferred to the router to re-enter the protocol.

Informative Findings

This section includes observations that are not directly exploitable but highlight areas for improvement in code clarity, maintainability, or best practices. While not critical, addressing these can strengthen the overall robustness of the system.

B01: Refactoring - Consistency when using `SafeERC20` library

Severity: Informative   Not addressed by client

In the `ZivoeVault` contract, the `SafeERC20` library is used inconsistently. Some function calls use:

runtimeverification/_audits_Zivoe_zivoe-vault/src/ZivoeVault.sol

Line 41 in 8efa4c9

```
        SafeERC20.safeTransferFrom(zSTT, caller, address(this), assets);
```

runtimeverification/_audits_Zivoe_zivoe-vault/src/ZivoeVault.sol

Line 56 in 8efa4c9

```
        SafeERC20.safeTransfer(zSTT, receiver, assets);
```

runtimeverification/_audits_Zivoe_zivoe-vault/src/ZivoeVault.sol

Line 77 in 8efa4c9

```
        SafeERC20.safeTransfer(asset, DAO, asset.balanceOf(address(this)));
```

Whereas in other places it is used as follows:

runtimeverification/_audits_Zivoe_zivoe-vault/src/ZivoeVault.sol

Line 43 in 8efa4c9

```
        zSTT.safeIncreaseAllowance(address(stSTT), assets);
```

runtimeverification/_audits_Zivoe_zivoe-vault/src/ZivoeVault.sol

Line 66 in 8efa4c9

```
            USD.safeIncreaseAllowance(address(ZVT), bal);
```

runtimeverification/_audits_Zivoe_zivoe-vault/src/ZivoeVault.sol

Line 69 in 8efa4c9

```
        zSTT.safeIncreaseAllowance(address(stSTT), bal2);
```

Recomendation

For consistency and improved readability, we recommend standardizing to a single form of invocation throughout the contract. This avoids confusion for developers maintaining the code and reduces the chance of subtle misuse.

B02: Increasing the `offset` protection

Severity: Informative | Not addressed by client

The OpenZeppelin implementation of `ERC4626` prevents against the inflation attack by using a virtual offset.

By default, the offset is set to 0. In this configuration, an inflation attack is non-profitable because the attacker's potential loss matches or exceeds the value of the user's deposit. According to the OpenZeppelin documentation, "if the offset is greater than 0, the attacker will have to suffer losses that are orders of magnitude bigger than the amount of value that can hypothetically be stolen from the user." Therefore, "bigger offsets increase the security even further by making any attack on the user extremely wasteful."

Consider increasing the offset if higher levels of security for this attack are intended (note that this also increases the number of decimals used to represent the shares in relation to the assets).

B03: `ZivoeRouter.depositWithPermit` can be more flexible

Severity: Informative | Not addressed by client

The function `depositWithPermit` requires that `msg.sender` be the permit signer. However, the function could be more flexible by allowing the sender to be different from the signer. This way, the signer can delegate to another address to make the deposit on its behalf.

Recomendation

```
    /// @param s The second 32 bytes of the permit signature.
    function depositWithPermit(address owner,  address stablecoin, uint256 amount, uint256
deadline, uint8 v, bytes32 r, bytes32 s)
        external
        nonReentrant
    {
        require(amount > 0, "Amount must be greater than zero");

        IERC20 USD = IERC20(stablecoin);

        // Use permit to approve USDC
        IERC20Permit(stablecoin).permit(owner, address(this), amount, deadline, v, r, s);

        // Continue with deposit logic
        USD.safeTransferFrom(owner, address(this), amount);
        USD.safeIncreaseAllowance(address(ZVT), amount);
        ZVT.depositSenior(amount, stablecoin);

        // Transfer ZVE if minted during depositSenior()
        uint256 balZVE = ZVE.balanceOf(address(this));
        if (balZVE > 0) {
            ZVE.safeTransfer(owner, balZVE);
        }

        // Deposit zSTT to vault (receiver is msg.sender)
        uint256 zSTT_bal = zSTT.balanceOf(address(this));
        zSTT.safeIncreaseAllowance(address(VLT), zSTT_bal);
        VLT.deposit(zSTT_bal, owner);
    }
```