Audit Report

Zodiac Protocol Core

v1.0

January 2, 2024

Table of Contents

License

Disclaimer

This audit has been performed by

Oak Security

https://oaksecurity.io/
info@oaksecurity.io

Introduction

Purpose of This Report

Oak Security has been engaged by Osmosis Grants Company to perform a security audit of Zodiac Protocol's Core contracts.

The objectives of the audit are as follows:

1. Determine the correct functioning of the protocol, in accordance with the project specification.
2. Determine possible vulnerabilities, which could be exploited by an attacker.
3. Determine smart contract bugs, which might lead to unexpected behavior.
4. Analyze whether best practices have been applied during development.
5. Make recommendations to improve code safety and readability.

This report represents a summary of the findings.

As with any code audit, there is a limit to which vulnerabilities can be found, and unexpected execution paths may still be possible. The author of this report does not guarantee complete coverage (see disclaimer).

Codebase Submitted for the Audit

The audit has been performed on the following target:

Repository	https://github.com/zodiac-protocol/contracts
Commit	`ce8348ca265258bb0e51cef70087a32cafeaa818`
Scope	All contracts were in scope.

Methodology

The audit has been performed in the following steps:

1. Gaining an understanding of the code base's intended purpose by reading the available documentation.
2. Automated source code and dependency analysis.
3. Manual line-by-line analysis of the source code for security vulnerabilities and use of best practice guidelines, including but not limited to:
    a. Race condition analysis
    b. Under-/overflow issues
    c. Key management vulnerabilities
4. Report preparation

Functionality Overview

Zodiac Protocol is a DeFi protocol that allows users to manage the risks associated with providing liquidity to AMMs (Automated Market Makers).
It deconstructs traditional LP tokens into Principal tokens, which remove the risk of volatile trading fees, and Yield tokens, which remove impermanent loss risk inherent to LPs.

How to Read This Report

This report classifies the issues found into the following severity categories:

Severity	Description
Critical	A serious and exploitable vulnerability that can lead to loss of funds, unrecoverable locked funds, or catastrophic denial of service.
Major	A vulnerability or bug that can affect the correct functioning of the system, lead to incorrect states or denial of service.
Minor	A violation of common best practices or incorrect usage of primitives, which may not currently have a major impact on security, but may do so in the future or introduce inefficiencies.
Informational	Comments and recommendations of design decisions or potential optimizations, that are not relevant to security. Their application may improve aspects, such as user experience or readability, but is not strictly necessary. This category may also include opinionated recommendations that the project team might not share.

The status of an issue can be one of the following: **Pending,** **Acknowledged**, or **Resolved**.

Note that audits are an important step to improving the security of smart contracts and can find many issues. However, auditing complex codebases has its limits and a remaining risk is present (see disclaimer).

Users of the system should exercise caution. In order to help with the evaluation of the remaining risk, we provide a measure of the following key indicators: **code complexity**, **code readability**, **level of documentation**, and **test coverage**. We include a table with these criteria below.

Note that high complexity or low test coverage does not necessarily equate to a higher risk, although certain bugs are more easily detected in unit testing than in a security audit and vice versa.

Code Quality Criteria

The auditor team assesses the codebase's code quality criteria as follows:

Criteria	Status	Comment
Code complexity	**High**	The protocol uses Stargate messages to communicate with the underlying Cosmos SDK appchain.
Code readability and clarity	**Low**	-
Level of documentation	**Medium**	-
Test coverage	**Medium-High**	`cargo tarpaulin` reports a test coverage for the contracts in scope of 90.97%.

Summary of Findings

No	Description	Severity	Status
1	Replaying `CloseHolder` message allows attackers to steal funds	Critical	Resolved
2	The vault contract is not able to mint YT tokens	Critical	Resolved
3	Incorrect vault expiration condition allows arbitrary closure of legitimate holders	Critical	Resolved
4	Large slippage tolerance allows for swap arbitraging	Major	Resolved
5	`FlashLoanV` message fails to transfer funds to the caller	Major	Resolved
6	Excess principal tokens will be stuck in the contract	Major	Resolved
7	Potential for front-running attack on `CloseVaultStageTwo` message	Major	Resolved
8	Holders with zero yield tokens cannot withdraw after stage two completion	Major	Resolved
9	Possibly outdated parent vault configurations may be used	Major	Resolved
10	Accrued fees are not sent to the fee collector contract	Major	Resolved
11	Non-callback arbitrary messages accepted by the `flash_loan_v` function	Major	Resolved
12	Misconfiguring fee values causes bank transfers to fail	Minor	Resolved
13	Vaults are not enforcing flash loan repayment and fees amount	Minor	Acknowledged
14	Incorrect error message in `calc_pt_yt_ownership_split` function	Minor	Resolved
15	Update of contract owner causes inconsistencies	Minor	Resolved
16	Update of vault configuration could lead to inconsistent state	Minor	Resolved
17	Possible vault key collision will lead to overwritten	Minor	Resolved

	data		
18	`RemoveTokenAction` transaction silently fails	Minor	Resolved
19	Contracts are not compliant with `CW2` Migration specification	Minor	Resolved
20	Owner can execute arbitrary `ComosMsgs`	Minor	Partially Resolved
21	Insufficient input validation across contracts	Minor	Partially Resolved
22	Inconsistent validation of assets	Minor	Acknowledged
23	Incorrect condition when querying flash-loaned amount	Informational	Resolved
24	Widespread usage of generic errors	Informational	Acknowledged
25	Incorrect error message	Informational	Resolved
26	Single-step ownership transfer	Informational	Acknowledged
27	Little attributes added to message handler responses	Informational	Acknowledged
28	Unused code within the contracts	Informational	Partially Resolved
29	Use of magic numbers decreases maintainability	Informational	Partially Resolved
30	“Migrate only if newer” pattern not followed	Informational	Acknowledged

Detailed Findings

1. Replaying `CloseHolder` message allows attackers to steal funds

Severity: Critical

In `contracts/osmosis/zodiac_osmo_bal_lockup_vault/src/contract.rs:657-658`, the `CloseHolder` message resets the holder's `unlocking_for_repaying` vector and distributes funds accordingly. However, the mutated holder information is not saved to the `YIELD_TOKEN_HOLDERS` storage, causing the storage to incorrectly reflect that the holder still has funds, despite them being already distributed.

Consequently, attackers can siphon funds from the vault by replaying the `CloseHolder` message with their own address, leading to a loss of funds for other holders. Moreover, if an attacker is closing their own position, they would receive all the funds except for the applied fee.

Please see the `test_repeated_holder_closure` test case to reproduce this issue.

Recommendation

We recommend saving the mutated holder information to the `YIELD_TOKEN_HOLDERS` storage.

Status: Resolved

2. The vault contract is not able to mint YT tokens

Severity: Critical

The `BlockBeforeSend` `Sudo` message, defined in `contracts/osmosis/zodiac_osmo_bal_vault/src/contract.rs:133`, is executed by the Osmosis chain when sending tokens through the `Bank` module and is used by the protocol to execute custom logic to map YT tokens to the recipient during a send operation.

This hook is implemented under the assumption that it is not executed when minting or burning tokens.

This assumption does not hold, however, since the mint operation calls the `SendCoinsFromModuleToAccount` function here which calls `SendCoins` here, which implies that the `BlockBeforeSend` hook is executed even during mint operations.

Since, during mint operations, the `fromAddr` is the address of the `tokenfactory` module, an error will be returned in

`contracts/osmosis/zodiac_osmo_bal_vault/src/contract.rs:149` when trying to decrease its zero balance.

Please see the `test_decrease_balance_user_not_existing` test case to reproduce this issue.

Recommendation

We recommend not executing the `BlockBeforeSend` hook logic during mint operations by implementing a guard to abort the execution if the sender is the `tokenfactory` module address.

Status: Resolved

3. Incorrect vault expiration condition allows arbitrary closure of legitimate holders

Severity: Critical

In `contracts/osmosis/zodiac_osmo_bal_lockup_vault/src/asserts.rs:132-134`, the `assert_holder_is_closeable` function attempts to check that the current time does not exceed the maturity timestamp when closing a holder. This condition is incorrect because holders should only be allowed to be closed after the vault has expired, implying they did not withdraw their locked tokens in time, which should result in a penalty.

Consequently, a malicious user could potentially close positions of holders who requested to unlock their yield tokens and receive the penalty amount as a reward, causing a loss of funds for the holder even if they intended to withdraw their locked tokens within the appropriate timeframe.

Recommendation

We recommend updating the condition in the `assert_holder_is_closeable` function only to allow closing holder accounts after the maturity timestamp.

Status: Resolved

4. Large slippage tolerance allows for swap arbitraging

Severity: Major

Slippage tolerance is used to determine the maximum deviation from the tokens that the user expects to receive to those actually received, for example, as a result of a swap. This is enforced through the `min_token_out` parameter. The smaller the value, the smaller amount of tokens can be received without the function returning an error.

The contracts within scope make use of very low `min_token_out` values for swaps, allowing other uses arbitrage profits by executing a sandwich attack.

This problem is exacerbated by the fact that swaps could be triggered by anonymous users through the `Keep` entrypoints. An attacker could monitor suitable target vaults in the protocol to arbitrage several swaps at once, choosing the time they are being executed by triggering the `Keep` functionality themselves.

The following instances use potentially exploitable limits:

- `contracts/osmosis/zodiac_osmo_bal_lockup_vault/src/contract.rs:299` and `804`
- `contracts/osmosis/zodiac_fee_collector/src/contract.rs:77` and `216`
- `contracts/osmosis/zodiac_osmo_vamm/src/contract.rs:301`
    - In this case, there are checks in place inside the reply handler that enforce a user-submitted slippage tolerance in line `329`. However, no validation is applied to that user-supplied value, and a default of zero min tokens is used, which could lead to a similar situation to the one described above.
- `contracts/osmosis/zodiac_osmo_vamm/src/contract.rs:453` and `552`
    - Similar to above, the user-supplied `token_out_lower_bound` is not validated for a minimum or suitable default.

Recommendation

We recommend enforcing both a suitable default and a suitable maximum value on slippage during swaps to restrict the economic losses to which the protocol and its users are exposed. We propose 5% as the default and 15% as the maximum slippage tolerance, given that a more conservative maximum such as 5% might cause issues with potentially low liquidity of the vaults at the beginning.

Status: Resolved

5. `FlashLoanV` message fails to transfer funds to the caller

Severity: Major

In `contracts/zodiac_flash_loan/src/contract.rs:196`, the `flash_loan_v` function is designed to enable the caller to execute a series of messages before repaying the flash loan. However, the function does not transfer the requested loan asset to the caller. Consequently, users or smart contracts intending to borrow funds using the `FlashLoanV` message will not receive the funds they expect.

Recommendation

We recommend transferring the requested loan asset to the caller, similar to the `flash_loan` function in line `184`.

Status: Resolved

6. Excess principal tokens will be stuck in the contract

Severity: Major

In `contracts/osmosis/zodiac_osmo_vamm/src/contract.rs:238-245`, the `TO_PLP_FLASH_LOAN_LOGIC_REPLY_ID` reply handler is designed to refund any excess yield tokens if the received principal tokens are less than required. However, there is no provision to handle a situation where the received principal tokens are more than the required yield tokens.

This scenario could occur if the pool for principal and liquidity tokens is imbalanced during the swap, as seen in lines `610-626`. Consequently, excess principal tokens will be left in the contract for anyone to retrieve.

Recommendation

We recommend refunding excess principal tokens to the swap caller.

Status: Resolved

7. Potential for front-running attack on `CloseVaultStageTwo` message

Severity: Major

In `contracts/osmosis/zodiac_osmo_bal_lockup_vault/src/contract.rs:634-640`, the `Keep` message is internally invoked during the vault's stage two closure process. The `keep` function implements a cooldown period in lines `829-831` such that it can only be called after a specific time.

As a result, malicious users can prevent others from calling the `CloseVaultStageTwo` message by frontrunning their transaction and calling the `keep` function first, causing it to enter the cooldown state, effectively blocking the stage two completion process.

We classify this issue as major because it affects the correct functioning of the system.

Recommendation

We recommend bypassing the cooldown period if the keep function is called internally.

Status: Resolved

8. Holders with zero yield tokens cannot withdraw after stage two completion

Severity: Major

In `contracts/osmosis/zodiac_osmo_bal_lockup_vault/src/contract.rs:440-453`, the withdrawal function attempts to send the holder's yield tokens once the vault's stage two is completed. This is problematic because it does not ensure that the holder's yield tokens are not zero. As Cosmos SDK prevents sending zero amounts, holders with zero yield tokens will be unable to withdraw their unlocked liquidity tokens, causing their funds to be locked in the contract.

We classify this issue as major because it affects the correct functioning of the system.

Recommendation

We recommend only sending yield tokens to holders if the locked amount is greater than zero.

Status: Resolved

9. Possibly outdated parent vault configurations may be used

Severity: Major

In `contracts/osmosis/zodiac_osmo_bal_lockup_vault/src/contract.rs:64-76`, the parent vault's configuration is queried and stored in the `Config` struct. If the parent vault configuration gets updated, the owner would need to initiate another transaction to reflect the new configuration in this contract, as seen in line `895`.

This approach leads to inconsistencies because an outdated parent vault configuration will be used if the owner does not update it immediately. Besides that, the owner can update the parent vault configuration to any arbitrary value, which might be incorrect.

We classify this issue as major because it affects the correct functioning of the system.

Recommendation

We recommend always performing a `VaultQueryMsg::Config` query to the parent vault address to retrieve the latest parent vault configuration, instead of storing the vault configuration in the child contract.

Status: Resolved

10. Accrued fees are not sent to the fee collector contract

Severity: Major

The flash loan contract calculates the loan's fee in the handler of the `FLASH_LOAN_REPLY_ID` reply in `contracts/zodiac_flash_loan/src/contract.rs:73` and `82`. However, instead of sending this amount to the `fee_collector` address as per the specification, it is sent to `snapshot.vault_address` alongside the actual loan.

As a consequence, the `CONFIG` field `fee_collector` is unused during the contract outside of query entry points.

Recommendation

We recommend following the specification by sending the calculated fee to the `fee_collector` address.

Status: Resolved

11. Non-callback arbitrary messages accepted by the `flash_loan_v` function

Severity: Major

The flash loan contract limits the `callback` messages that are executed through the `flash_loan` function to `ExecuteMsg` directed to the sender only, as best practices dictate. However, this is not the case for the additional entry point handled by the `flash_loan_v` function. In that case, the function accepts arbitrary Cosmos messages for execution without any further restriction.

For instance, this message could be used to move the contract's funds to an arbitrary address through a `BankMsg`. This would allow anyone to drain all the contract's funds when asking for a loan through the affected functionality.

We classify this issue as major instead of critical, given that the `flash_loan` contract is not expected to hold any permissions on other contracts or hold any funds.

Recommendation

We recommend restricting the valid `callback` messages to `ExecuteMsg` directed towards the sender, as done in the `flash_loan` function.

Status: Resolved

12. Misconfiguring fee values causes bank transfers to fail

Severity: Minor

In several instances of the contract, amounts to be transferred are not checked to be greater than zero. For instance, if the contract owner misconfigures the fee value to be zero or the max value of `10000`, either the fee or the actual transfer will fail and the transaction will revert since Cosmos SDK prevents sending zero amounts.

The following code lines are affected:

- `contracts/osmosis/zodiac_osmo_bal_lockup_vault/src/contract.rs:419`, `427`, `568`, `578`, `694`, `704`, `728`, and `736`.
- `contracts/osmosis/zodiac_osmo_bal_vault/src/contract.rs:718` and `719`.

We classify this issue as minor because the contract owner can recover from it by updating the fee configuration.

Recommendation

We recommend performing a transfer of funds only if the amount is greater than zero.

Status: Resolved

13. Vaults are not enforcing flash loan repayment and fees amount

Severity: Minor

Vaults are designed not to enforce any specific logic for flash loans, like the requested fee amount and the restriction to repay the loan in the same block.

Those functionalities are implemented in the flash loan contract, and the vault owner can allow a trusted flash loan contract to operate its funds.

While this approach is achieving a good level of modularity, it shifts safety checks from the vault to the flash loan contract, which increases the attack surface.

A compromised flash loan contract could allow an attacker to disrupt the service for vault users by blocking withdrawals or draining funds from the vault. This could happen, for

example, if the keys of the flash loan contract's admin are compromised, allowing an attacker to upgrade the contract.

We classify this issue as minor because the flash loan contract is a trusted entity set by the vault owner.

Recommendation

We recommend implementing the requirement to repay the loan in the same block and requesting a fixed fee for flash loans directly in the vault contract.

Status: Acknowledged

14. Incorrect error message in `calc_pt_yt_ownership_split` function

Severity: Minor

In `contracts/osmosis/zodiac_osmo_bal_lockup_vault/src/utils.rs:123-128`, the error message states `"at sqrt(k) {lp_per_l} - compose of {pool_l} / {pool_shares}"`, indicating that `lp_per_l` is calculated from `pool_l` divided by `pool_shares`.

This message is incorrect because `lp_per_l` is calculated from `pool_shares` divided by `pool_l`, as seen in line `111`.

Recommendation

We recommend correcting the error message.

Status: Resolved

15. Update of contract owner causes inconsistencies

Severity: Minor

In `contracts/osmosis/zodiac_osmo_bal_vault/src/contract.rs:358` and `contracts/osmosis/zodiac_osmo_factory/src/contract.rs:165`, the `CreateYieldVault` message and `create_vault` function, respectively, instantiate a contract with the admin set to `config.owner`. This could be an issue since `config.owner` is an updatable field.

Updating the `owner` of these contracts leads to an inconsistency where the contract's admin differs from the current contract `owner`. Consequently, the previous `owner` may still have the

capability to migrate the contracts, even though they are no longer the current contract `owner`.

Recommendation

We recommend performing a `MsgUpdateAdmin` for instantiated contracts upon an `owner` update to transfer the contract admin to the new `owner`.

Status: Resolved

16. Update of vault configuration could lead to inconsistent state

Severity: Minor

In `contracts/osmosis/zodiac_osmo_bal_vault/src/contract.rs:577` and `contracts/osmosis/zodiac_osmo_factory/src/contract.rs:581`, the `owner` is able to update the `maturity_timestamp` and the `collateral_token`, respectively.

Updating these values leads to inconsistency if the contracts are already in use. Changing the maturity timestamp would manipulate the market since it will indirectly affect the value of `PT` and `YT` tokens. Updating the vault's collateral token leads to all deposits and stored data referring to the wrong token.

Recommendation

We recommend not allowing the `owner` to update the `maturity_timestamp` or the `collateral_token`.

Status: Resolved

17. Possible vault key collision will lead to overwritten data

Severity: Minor

In `contracts/osmosis/zodiac_osmo_factory/src/contract.rs:187`, during vault instantiation, a key is generated and stored.

The key is composed of `vault_type` and `collateral_token`. However, no separator is used. Consequently, key collisions are possible with values that have the same concatenated result. If such a key collision occurs, existing data will be overwritten.

We classify this issue as minor because the `vault_type` and `collateral_token` values are provided by the owner.

Please see the `test_create_vault_key_collision` test case to reproduce the issue.

Recommendation

We recommend adding a separator between `vault_type` and `collateral_token` that cannot be part of either value.

Status: Resolved

18. `RemoveTokenAction` message silently fails

Severity: Minor

The `remove_token_action` function, defined in `contracts/osmosis/zodiac_fee_collector/src/contract.rs:135`, silently fails if the provided `token_action` denom is not stored in the contract.

Consequently, a contract that executes the `RemoveTokenAction` message cannot correctly handle this error, and users are not getting feedback on the execution status.

Recommendation

We recommend reverting the transaction when the execution is not able to remove a `token_action`.

Status: Resolved

19. Contracts are not compliant with `CW2` Migration specification

Severity: Minor

The protocol contracts do not adhere to the `CW2` Migration specification standard. This may lead to unexpected problems during contract migration and code version handling.

Recommendation

We recommend following the CW2 standard in all the contracts. For reference, see https://docs.cosmwasm.com/docs/1.0/smart-contracts/migration.

Status: Resolved

20. Owner can execute arbitrary `ComosMsgs`

Severity: Minor

The `vault` contract allows the `owner` to execute arbitrary CosmosMsgs through the `ExecuteMsg::OwnerAction` entry point.

Among others, this message can be used to move contract funds to an arbitrary address through a `BankMsg`. In the event of compromised access keys or a malicious insider, this would allow the sweeping of all the funds of the contract.

In addition, the `fee collector` contract allows callbacks to execute arbitrary CosmosMsgs through the `ExecuteMsg::Execute` entry point in `contracts/osmosis/zodiac_fee_collector/src/contract.rs:120-127`. As this callback can be executed as part of the `CONFIG.token_actions` defined by the owner, they will be able to select the arbitrary message that would be executed.

Recommendation

We recommend restricting the allowed messages to the minimum subset that will be needed for the protocol's operation.

Status: Partially Resolved

21. Insufficient input validation across contracts

Severity: Minor

The smart contracts within the scope currently lack sufficient validation before saving configuration details. This could lead to issues that disrupt the correct behavior of the protocol and possibly lead to failing transactions.

Firstly, the `lockup vault` contract misses validation upon instantiation and update in `contracts/osmosis/zodiac_osmo_bal_lockup_vault/src/contract.rs:73-84, 752-773, 879-906`:

- `token_actions`:
  - Lack of `denom` validation could lead to failing transactions.
  - Duplicate `order` value could cause undesired behavior when processing transactions.
  - Line `858` states that the message should be restricted to `executmsg::swapdenom{in_denom, out_denom, pool}`. If the only expected messages are the contract-only entry points `SwapDenom` and `SwapToPool`, the message should be restricted accordingly.
  - If the action in the `KeeperTokenAction` is a `SwapToPool` message, the `pool_id` should be validated to hold the LP token denom as the parent vault's `collateral_token` denom to prevent incorrect LP tokens accruing in lines `227-229`.
  - If the action in the `KeeperTokenAction` is a `SwapDenoms` message, the `denom_in` should be validated not to be the yield token denom to prevent transferring them.
  - A max limit should be implemented to the number of token actions to prevent an out-of-gas scenario in lines `848-864`.

    - Allowed messages in the `KeeperTokenAction` should be limited to `SwapDenoms` and `SwapToPool` because other messages would fail.
- `maturity_timestamp` is not enforced to be in the future, which could disrupt the expected behavior of the protocol. This happens during the contract instantiation and configuration update.
- `lockup_duration` should be larger than zero, as `MsgLockTokens` of zero has no effect.
- `keep_cooldown_seconds` should be larger than zero to implement a minimum cooldown duration.
- `closing_penalty_bps` is not checked to be within the 0-9999 range. Fee transfer messages will fail if equal to the minimum, and the amount affected by the fee will fail to be transferred if a maximum fee is applied.

Secondly, the `fee collector` contract misses validation upon instantiation and update in `contracts/osmosis/zodiac_fee_collector/src/contract.rs:17-32` and `157-178`:

- `token_actions`:
    - Lack of `denom` validation could lead to failing transactions.
    - Repeated `order` could cause undesired behavior when sending transactions.
    - A max limit should be implemented to the number of token actions to prevent an out-of-gas scenario in lines `246-260`.
    - Possible centralization risk due to arbitrary Cosmos messages via `ExecuteMsg::Execute` message.

Thirdly, the `vault` contract misses validation upon instantiation and configuration update in `contracts/osmosis/zodiac_osmo_bal_vault/src/contract.rs:559-595`:

- `maturity_timestamp` should not be in the past for the protocol to work as intended. Besides that, this should not be updatable to prevent state inconsistencies between the factory contract and the vault contract.
- `collateral_token` should be checked with `pool_id` to ensure the pool indeed uses the provided collateral token denom as liquidity token. Besides that, the value should not be updatable to prevent state inconsistency regarding contract balance.
- `ptoken_l` should be checked to not be zero to prevent a division by zero error in line `112`.
- `pool_id` should be checked to ensure the pool is a XYK pool with a total of two assets.
- `claim_yield_fee` is not checked to be within the 0-9999 range.
- `redeem_fee` is not checked to be within the 0-10000 range.

Fourthly, the `factory` contract does not validate the `vault_types` upon instantiation or update in `contracts/osmosis/zodiac_osmo_factory/src/contract.rs:29-43` and `92-94`. It should be noted that identifiers of type string are allowed for this field, which is prone to errors compared to an `enum`.

Lastly, the `flash loan` contract does not validate the `fee` to be between 0 and 10000 upon instantiation and update in `contracts/zodiac_flash_loan/src/contract.rs:28-47` and `247-275`.

Recommendation

We recommend thoroughly validating all the affected parameters.

Status: Partially Resolved

22. Inconsistent validation of assets

Severity: Minor

The `flash loan` contract enforces a requirement on `contracts/zodiac_flash_loan/src/contract.rs:99` for the `loan_asset` to be a native asset. However, this condition is not enforced on the `flash_loan` and `flash_loan_v` functions.

Recommendation

We recommend consistently enforcing any restriction on `loan_asset` throughout the codebase.

Status: Acknowledged

23. Incorrect condition when querying flash-loaned amount

Severity: Informational

In `contracts/osmosis/zodiac_osmo_vamm/src/contract.rs:489-491`, the guard verifies whether the contract balance is greater than the requested borrow amount. This is incorrect because it does not include the initially provided yield tokens, which are swapped into liquidity tokens.

Recommendation

We recommend checking whether the contract balance is lower than `to_ylp_swapper_info.alpha` and `to_ylp_swapper_info.borrow_amount` and return an error.

Status: Resolved

24. Widespread usage of generic errors

Severity: Informational

The contracts within the scope of this audit make use of generic errors instead of defining custom errors. Although not a security issue, this approach reduces the readability and maintainability of the project.

Recommendation

We recommend defining custom errors and using them consistently throughout the codebase.

Status: Acknowledged

25. Incorrect error message

Severity: Informational

The `fee collector` contract returns an incorrect error message in `contracts/osmosis/zodiac_fee_collector/src/contract.rs:123`. This error message wrongly states that the contract is a yield vault instead of the fee collector, which could be misleading for users.

Recommendation

We recommend correcting this error message.

Status: Resolved

26. Single-step ownership transfer

Severity: Informational

The contracts within the scope of this audit allow the current owner to execute a one-step ownership transfer. While this is common practice, it presents a risk for the ownership of the contract to become lost if the owner transfers ownership to the incorrect address. A two-step ownership transfer will allow the current owner to propose a new owner, and then the account that is proposed as the new owner may call a function that will allow them to claim ownership and actually execute the config update.

Recommendation

We recommend implementing a two-step ownership transfer. The flow can be as follows:

- The current owner proposes a new owner address that is validated and lowercase.
- The new owner account claims ownership, which applies the configuration changes.

Status: Acknowledged

27. Little attributes added to message handler responses

Severity: Informational

The contracts within scope rarely make use of attributes when returning a response, either at the end of the execution or at an early exit. This could negatively impact off-chain services that try to monitor the state of the protocol.

Recommendation

We recommend adding enough information as attributes to responses so the performed action and outcome can be clearly identified by off-chain services.

Status: Acknowledged

28. Unused code within the codebase

Severity: Informational

The codebase contains several declarations, both variables and functions, that are not used anywhere except in debugging code. Although not a security issue, unused code goes against best practices since it decreases maintainability and readability.

Instances of unused code can be found in:

- `contracts/osmosis/zodiac_osmo_bal_lockup_vault/src/contract.rs:35, 36, 37`
- `contracts/osmosis/zodiac_osmo_vamm/src/math.rs:55-58`
- `contracts/osmosis/zodiac_osmo_vamm/src/contract.rs:77, 86-90, 133`

In addition, the validation in `contracts/osmosis/zodiac_osmo_factory/src/contract.rs:148-150` is redundant as line `146` would already return an error if an invalid `vault_type` is supplied.

Recommendation

We recommend removing any unused code.

Status: Partially Resolved

29. Use of magic numbers decreases maintainability

Severity: Informational

Throughout the codebase, hard-coded number literals without context or a description are used. Using such "magic numbers" goes against best practices as they reduce code readability and maintenance as developers are unable to easily understand their use and may make inconsistent changes across the codebase.

Instances of magic numbers are listed below:

- `contracts/osmosis/zodiac_osmo_bal_lockup_vault/src/contract.rs:409, 548, 686, 718`
- `contracts/osmosis/zodiac_osmo_bal_lockup_vault/src/contract.rs:708, 722`
- `contracts/osmosis/zodiac_osmo_factory/src/contract.rs:152`
- `contracts/osmosis/zodiac_osmo_vamm/src/contract.rs:184, 279, 458, 557`
- `contracts/osmosis/zodiac_osmo_vamm/src/math.rs:19, 21`
- `contracts/zodiac_flash_loan/src/contract.rs:56, 575`

Recommendation

We recommend defining magic numbers as constants with descriptive variable names and comments, where necessary.

Status: Partially Resolved

30. "Migrate only if newer" pattern not followed

Severity: Informational

The contracts within the scope of this audit are currently migrated without regard to their version. This can be improved by adding validation to ensure that the migration is only performed if the supplied version is newer.

Recommendation

It is recommended to follow the migrate “only if newer” pattern defined in the CosmWasm documentation.

Status: Acknowledged

Appendix: Test Cases

1. Test case for "Replaying `CloseHolder` message allows attackers to steal funds"

```
#[test]
    fn test_repeated_holder_closure() {
        let mut mock_deps = zodiac_mocks::mock_dependencies(&[]);
        let mock_env =
zodiac_mocks::mock_env(zodiac_mocks::MockEnvParams::default());
        clean_setup(&mut mock_deps);

        // create user
        let random_user = "random";
        let mock_info = zodiac_mocks::mock_info(&String::from(random_user));

        // setup
        let mut closeable_holder = Holder {
            locked_yt: Uint128::new(0),
            locked_lp: Uint128::new(0),
            unlocking_for_repaying: vec![],
            unlocking_for_claiming: vec![],
            index: Decimal::zero(),
            accrued_yield: Decimal::zero(),
        };

        closeable_holder.unlocking_for_repaying = vec![(
            Uint128::new(100_000),
            Uint128::new(10_000_000_000_000_000_000),
            mock_env.block.time.seconds(),
        )];

        YIELD_TOKEN_HOLDERS
            .save(
                &mut mock_deps.storage,
                &Addr::unchecked(random_user),
                &closeable_holder,
            )
            .unwrap();

        // query holder info before closing
        let holder_response: Holder = from_binary::<Holder>(
            &query(
                mock_deps.as_ref(),
                mock_env.clone(),
                QueryMsg::Holder {
                    address: random_user.to_string(),
                },
```

```
            )
            .unwrap(),
        )
        .unwrap();

        // close holder
        execute(
            mock_deps.as_mut(),
            mock_env.clone(),
            mock_info,
            ExecuteMsg::CloseHolder {
                address: String::from(random_user),
            },
        )
        .unwrap();

        // query after close holder
        let post_holder_response: Holder = from_binary::<Holder>(
            &query(
                mock_deps.as_ref(),
                mock_env,
                QueryMsg::Holder {
                    address: random_user.to_string(),
                },
            )
            .unwrap(),
        )
        .unwrap();

        // no change in state
        assert_eq!(holder_response, post_holder_response);

    }
```

2. Test case for “The vault contract is not able to mint YT tokens”

```
#[test]
    fn test_decrease_balance_user_not_existing() {
        let mut mock_deps = zodiac_mock_shit::mock_dependencies(&[]);
        mock_deps.querier.set_query_pool_response(69u64, None);

        CONFIG.save(&mut mock_deps.storage, &Config{
            owner: Addr::unchecked("owner"),
            collateral_token: String::from("gamm/69"),
            maturity_timestamp: (1571797419u64 + 60u64*60u64*24u64*30u64),
            principal_token: String::from("factory/owner/p"),
            yield_token: String::from("factory/owner/y"),
            pool_id: 69u64,
            ptoken_1: Uint128::one(), //liquidity per ptoken
            flash_loan_address: Addr::unchecked("flash_loan"),
            fee_collector: Addr::unchecked("fee_collector"),

            //fee configs, all in bps (all denominated in collateral token)
            redeem_fee: 30u64,
            claim_yield_fee: 1000u64,

            display_name: String::from("a-b"),
            yield_token_hook_contract: None,
        }).unwrap();

        STATE.save(&mut mock_deps.storage, &State{
            global_index: Decimal::zero(),  //this is cumlative_sum(yield / yield_token_supply); is an intermediate calc for ytoken holder bookkeeping
            previous_balance: Uint128::zero(), // this is the LP balance earmarked for ytoken holders
            collateral_balance: Uint128::from(1000u128), //track collateral balance of this vault to save query calls + maintain balance state for yt accounting during flash loans
            amount_flash_loaned_out: Uint128::zero(),

            flash_loan_enabled: true,
        }).unwrap();

        YIELD_VAULTS.save(&mut mock_deps.storage, &vec![]).unwrap();

        let state: State = STATE.load(&mut mock_deps.storage).unwrap();
        decrease_balance(&mock_deps.as_ref(), &state, &String::from("tokenfactorymodule"), Uint128::from(20u32)).unwrap();
    }
```

3. Test case for [“Possible vault key collision will lead to overwritten data”]

```
#[test]
    fn test_create_vault_key_collision(){
        let mut mock_deps = zodiac_mocks::mock_dependencies(&[]);
        let mock_env =
zodiac_mocks::mock_env(zodiac_mocks::MockEnvParams::default());
        let mock_info = zodiac_mocks::mock_info(&String::from("zodiac"));
        clean_setup(&mut mock_deps);

        instantiate(mock_deps.as_mut(), mock_env.clone(), mock_info,
InstantiateMsg{
            vault_types: vec![String::from("balancer")]
        }).unwrap();

        let config: Config = CONFIG.load(&mut mock_deps.storage).unwrap();
        assert_eq!(config.vault_types, vec![String::from("balancer")]);
        assert_eq!(config.owner, Addr::unchecked("zodiac"));

        let mock_info = zodiac_mocks::mock_info(&String::from("zodiac"));
        execute(mock_deps.as_mut(), mock_env.clone(), mock_info.clone(),
ExecuteMsg::UpdateConfig{
            owner: Some(String::from("asdf")),
            vault_types: Some(vec![String::from("balancer"),
String::from("balancera"), String::from("solidly")]),
        }).unwrap();

        let config_response: ConfigResponse =
from_binary::<ConfigResponse>(&query(mock_deps.as_ref(), mock_env.clone(),
QueryMsg::Config{}).unwrap()).unwrap();

        let config: Config = CONFIG.load(&mut mock_deps.storage).unwrap();
        assert_eq!(config_response.owner, config.owner);

        let config: Config = CONFIG.load(&mut mock_deps.storage).unwrap();
        assert_eq!(config.vault_types, vec![String::from("balancer"),
String::from("balancera"), String::from("solidly")]);
        assert_eq!(config.owner, Addr::unchecked("asdf"));

        let mock_info = zodiac_mocks::mock_info(&String::from("asdf"));
        execute(mock_deps.as_mut(), mock_env.clone(), mock_info.clone(),
ExecuteMsg::UpdateVaultConfig{
            config: VaultConfig{
                code_id: 29u64,
                vault_type: String::from("balancer"),
            }
        }).unwrap();

        execute(mock_deps.as_mut(), mock_env.clone(), mock_info.clone(),
```

```
ExecuteMsg::UpdateVaultConfig{
            config: VaultConfig{
                code_id: 29u64,
                vault_type: String::from("balancera"),
            }
        }).unwrap();

        let vault_config: VaultConfig = VAULT_CONFIGS.load(&mut
mock_deps.storage, String::from("balancer")).unwrap();
        assert_eq!(vault_config.code_id, 29u64);
        assert_eq!(vault_config.vault_type, String::from("balancer"));

        let maturity_year = 2023i64;
        let maturity_month = 12u64;


        execute(mock_deps.as_mut(), mock_env.clone(), mock_info.clone(),
ExecuteMsg::CreateVault{
            vault_type: String::from("balancer"),
            collateral_token: String::from("asdf"),
            maturity_month,
            maturity_year,
            options: None,
            name: Some(String::from("ddd")),
            owner: mock_info.sender.to_string()
        }).unwrap();

        let created_vault_first = vault_key( &CREATING_VAULT.load(&mut
mock_deps.storage).unwrap());

        execute(mock_deps.as_mut(), mock_env.clone(), mock_info.clone(),
ExecuteMsg::CreateVault{
            vault_type: String::from("balancera"),
            collateral_token: String::from("sdf"),
            maturity_month,
            maturity_year,
            options: None,
            name: Some(String::from("eee")),
            owner: mock_info.sender.to_string()
        }).unwrap();

        let created_vault_second = vault_key( &CREATING_VAULT.load(&mut
mock_deps.storage).unwrap());

        assert_ne!(created_vault_first, created_vault_second);


    }
```