Audit Report

Frizzante Core Contracts

v1.0

January 30, 2025

Table of Contents

License

Disclaimer

THE CONTENT OF THIS AUDIT REPORT IS PROVIDED “AS IS”, WITHOUT REPRESENTATIONS AND WARRANTIES OF ANY KIND.

THE AUTHOR AND HIS EMPLOYER DISCLAIM ANY LIABILITY FOR DAMAGE ARISING OUT OF, OR IN CONNECTION WITH, THIS AUDIT REPORT.

THIS AUDIT REPORT WAS PREPARED EXCLUSIVELY FOR AND IN THE INTEREST OF THE CLIENT AND SHALL NOT CONSTRUE ANY LEGAL RELATIONSHIP TOWARDS THIRD PARTIES. IN PARTICULAR, THE AUTHOR AND HIS EMPLOYER UNDERTAKE NO LIABILITY OR RESPONSIBILITY TOWARDS THIRD PARTIES AND PROVIDE NO WARRANTIES REGARDING THE FACTUAL ACCURACY OR COMPLETENESS OF THE AUDIT REPORT.

FOR THE AVOIDANCE OF DOUBT, NOTHING CONTAINED IN THIS AUDIT REPORT SHALL BE CONSTRUED TO IMPOSE ADDITIONAL OBLIGATIONS ON COMPANY, INCLUDING WITHOUT LIMITATION WARRANTIES OR LIABILITIES.

This audit has been performed by

Oak Security GmbH

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 Frizzante 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/frizzante-finance/core
Commit	`566319a84653858d9aa68a045b1d88c84a76e1a9`
Scope	The scope was restricted to the following directories: • `contracts/notional-pool` • `contracts/yield-token` • `contracts/router` • `packages/balances` • `packages/lazy-distribution` • `packages/proto` • `packages/storage` • `packages/sy`

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

Frizzante is a DeFi protocol that allows users to tokenize yield-bearing assets into their principal and yield components as different tokens, enabling a variety of trading strategies. The principal token can be redeemed fully for the accounting asset after maturity, while the yield tokens distribute accrued interest to the holders.

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 codebase implements complex reward distribution mechanisms for yield tokens and AMM trading mechanisms for standardized yield and principal tokens. Additionally, complex cross-contract calls and reply callbacks are implemented.
Code readability and clarity	**Low-Medium**	There are multiple compiler warnings (e.g., deprecated instances, unused parameters) and TODO comments in the codebase.
Level of documentation	**Medium-High**	Documentation is available at `contracts/README.md`.
Test coverage	**Medium-High**	`cargo tarpaulin` reports a test coverage of `85.68%`.

Summary of Findings

No	Description	Severity	Status
1	Incorrect calculation of user shares leads to disproportionate rewards	Critical	Resolved
2	Shares are minted to the incorrect address	Critical	Resolved
3	User rewards are not accrued before burning yield tokens	Critical	Resolved
4	Missing division by zero validation allows attackers to break the standardized yield contract	Critical	Resolved
5	Inflated standardized yield token balance causes incorrect reward distribution	Critical	Resolved
6	Attackers can cause reward updates to fail due to excessive iteration	Critical	Resolved
7	Liquidity withdrawal may fail due to the sensitivity of the logit curve at its edges	Major	Resolved
8	Unvalidated treasury fee value	Minor	Acknowledged
9	Incomplete derivation of interest rate formula	Informational	Acknowledged

Detailed Findings

1. Incorrect calculation of user shares leads to disproportionate rewards

Severity: Critical

In `contracts/yield-token/src/rewards.rs:44-46`, the `asset_to_sy` function incorrectly converts users' yield token balances into standardized yield tokens by multiplying them with the exchange rate instead of dividing them. This flawed conversion method results in an overestimation of user shares, effectively rewarding users with more rewards than intended.

Consequently, early adopters may claim an excess share of rewards, preventing subsequent users from receiving their intended share due to an insufficient balance error.

Recommendation

We recommend computing the standardized yield token balance by dividing it by the exchange rate, similar to `contracts/yield-token/src/burn.rs:77`.

Status: Resolved

2. Shares are minted to the incorrect address

Severity: Critical

In `packages/sy/src/contract.rs:187`, the `deposit` function sets the receiver of the shares to the caller's address, indicating the shares will be minted to the caller in line `377`. This is incorrect because a receiver parameter is implemented in line `171`, which should be the recipient instead of the caller.

Recommendation

We recommend setting the recipient to the receiver address in `packages/sy/src/contract.rs:173`.

Status: Resolved

3. User rewards are not accrued before burning yield tokens

Severity: Critical

In `contracts/yield-token/src/burn.rs:29`, the `burn` function reduces the user's yield token balance when burning principal and yield tokens in exchange for standardized yield tokens.

The issue is that the user's rewards index was not updated before the yield token balance was decreased. This is important because the yield token balance determines the rewards the user should receive, as seen in `contracts/yield-token/src/interest.rs:91-94`.

Consequently, users will receive fewer rewards than intended as their rewards will be computed with the reduced yield token balance, causing a loss of yield for the user and the undistributed yield to be stuck in the contract.

Recommendation

We recommend calling the `calculate_accrued_rewards_and_interest` function to accrue user rewards before decreasing their yield token balance.

Status: Resolved

4. Missing division by zero validation allows attackers to break the standardized yield contract

Severity: Critical

In `packages/lazy-distribution/src/lib.rs:184`, the `update_indexes` function divides the amount of the rewards with the total shares. This is problematic because if no shares are minted, a division by zero error will occur, reverting the transaction.

For example, if an attacker sends a reward token to the standardized yield contract before any SY tokens are minted, a division by zero error will occur when computing the rewards in `packages/sy/src/contract.rs:162`. Since the standardized yield contract updates the rewards before dispatching other actions (e.g., deposit and redemption), the error will always be triggered, causing a denial of service.

This issue also affects the notional pool and yield token contracts because they use the affected code to distribute rewards.

Please refer to the `test_dos_sy_contract` test case in the appendix to reproduce this issue.

Recommendation

We recommend updating the reward index only if the total share value exceeds zero.

Status: Resolved

5. Inflated standardized yield token balance causes incorrect reward distribution

Severity: Critical

In `packages/lazy-distribution/src/lib.rs:165`, the `update_indexes` function computes the total shares and distributes the rewards among stakers in line `184`. In the case of the yield token contract, the total shares are represented as the standardized yield (SY) token balance owned by the contract, as seen in `contracts/yield-token/src/rewards.rs:26`.

The issue is that when users send SY tokens through the `Receive` entry point in `contracts/yield-token/src/contract.rs:282-289`, the total share value will be inflated due to the increased SY token balance. This is problematic because rewards should only be allocated to stakers who have staked during the reward accrual period, and the newly sent SY tokens should be excluded from receiving rewards.

Consequently, legitimate stakers will receive fewer rewards than intended, and the undistributed rewards will be stuck in the yield token contract.

Please refer to the `test_stuck_rewards` test case in the appendix to reproduce this issue.

Recommendation

We recommend modifying the `get_total_shares` function in `contracts/yield-token/src/rewards.rs:26` to exclude the sent standardized yield tokens.

Status: Resolved

6. Attackers can cause reward updates to fail due to excessive iteration

Severity: Critical

In `packages/lazy-distribution/src/lib.rs:137`, the `before_users_share_change` function iterates over all global reward indexes to compute the users' rewards. The global reward indexes are added when new reward tokens are sent to the standardized yield contract in

`packages/sy/src/contract.rs:151-162`, which will eventually be distributed to stakers.

This approach is problematic because an attacker can send many different reward tokens to cause an out-of-gas error when iterating over all reward tokens. An attacker can create worthless reward tokens via the `x/tokenfactory` module at a low cost.

Consequently, the standardized contract may fail to accrue the rewards, causing a denial of service that also affects the notional pool and yield token contracts, breaking the protocol's functionality.

This issue also affects the notional pool, as an attacker can trigger a denial-of-service by sending many different tokens, see `contracts/notional-pool/src/contract.rs:157`.

Recommendation

We recommend modifying the implementation to introduce a whitelist mechanism for reward tokens. This approach ensures that only tokens in the whitelist can be accrued as rewards for stakers, which can be configured by the contract owner.

Status: Resolved

7. Liquidity withdrawal may fail due to the sensitivity of the logit curve at its edges

Severity: Major

In `contracts/notional-pool/src/pool/mod.rs:210-212`, the `remove_liquidity` function returns an error if the standardized yield or principal token is zero (`net_sy_to_account.is_zero() || net_pt_to_account.is_zero()`). This is problematic as logit curves, particularly at their edges, exhibit sensitivity to input values, often leading to precision loss.

Due to this, the return amount of standardized yield or principal token representing the asset quantities may become zero, especially when a rounding issue occurs, which is an expected behavior.

Consequently, users cannot withdraw liquidity, causing funds to be locked.

Recommendation

We recommend modifying the implementation to use `net_sy_to_account.is_zero() && net_pt_to_account.is_zero()`.

Status: Resolved

8. Unvalidated treasury fee value

Severity: Minor

In `contracts/yield-token/src/instantiate.rs:49`, the instantiate function does not validate that the `TREASURY_FEE` is less than 100%. If it is misconfigured, the rewards cannot be updated due to an overflow error in `contracts/yield-token/src/interest.rs:41-43`.

We classify this issue as minor because it can only be caused by the contract instantiator, which is a privileged role.

Recommendation

We recommend validating that the fee is less than 100%, i.e., `Decimal::one`.

Status: Acknowledged

9. Incomplete derivation of interest rate formula

Severity: Informational

The following derivation can be found In `contracts/yield-token/src/interest.rs:65-76`:

```
// interest_sys = {[(principal / old_rate) *
current_rate]-total_supply}/current_rate
// Which simplifies to:
principal.mul_floor(current_rate - old_rate).div_floor(old_rate * current_rate)
```

However, the equation

$$interest_{sys} = \frac{\frac{principal}{rate_{old}} rate_{current} - supply_{total}}{rate_{current}}$$

simplifies to

$$interest_{sys} = principal/rate_{old} - supply_{total}/rate_{current}$$

and only with the additional assumption that $principal = supply_{total}$ it would simplify to

$$interest_{sys} = principal \frac{rate_{current} - rate_{old}}{rate_{old}\ rate_{current}}$$

The issue is that the principal within the computations is not guaranteed to equal the total supply as the `calc_sy_interest` function is called with `total_supply - PT_DIVERGENCE` as input parameters in `contracts/yield-token/src/interest.rs:56`.

We classify this issue as informational as the current implementation is correct, but the documentation for the calculation is incomplete.

Recommendation

We recommend documenting the mathematical derivations completely.

Status: Acknowledged

Summary of Findings For Router Contract

No	Description	Severity	Status
10	Fees sent to the router contract can be stolen	Critical	Resolved
11	Unused tokens are not refunded to the caller	Critical	Acknowledged
12	`ListMarkets` query fails if any market's principal token is expired	Minor	Resolved
13	Incorrect contract label configured for the yield token	Informational	Resolved
14	Incorrect `SwapExactSyForYtResponse.net_yt_out` field due to rounding issue	Informational	Acknowledged

Detailed Findings For Router Contract

10. Fees sent to the router contract can be stolen

Severity: Critical

In `contracts/router/src/admin.rs:110` and `contracts/router/src/admin.rs:161`, the router contract instantiates the yield token and notional pool contract, which sets both contract's treasury address to the router contract, as seen in `contracts/yield-token/src/instantiate.rs:47` and `contracts/notional-pool/src/contract.rs:51`. This means that the distributed fees will be sent to the router contract.

However, this approach is problematic because the router contract is not supposed to hold any funds, as anyone can steal them. For example, the standardized yield tokens in the router contract can be stolen by utilizing the `JoinDual` message and receiving them as liquidity pool tokens.

Consequently, standardized yield, principal, and yield tokens distributed to the router contract may be stolen, causing the protocol to lose fee income.

Recommendation

We recommend modifying the instantiation logic for the yield token and notional pool contracts so the treasury is set to another privileged address, such as the contract admin address.

Status: Resolved

11. Unused tokens are not refunded to the caller

Severity: Critical

The router contract is not intended to hold funds after a successful operation, as these funds can be stolen by calling messages that do not validate the sent funds amount (e.g., `JoinDual` and `SwapExactPtForSy` messages). Since no refund mechanism is automatically executed after any operation, any unused tokens may be lost.

Firstly, suppose the user sends the principal token when calling the `JoinSingleSyCoin`, `JoinSingleSyCoinKeepYt`, `SwapExactSyCoinForPt`, and `SwapExactSyCoinForYt` messages. In that case, these tokens are excluded when depositing into the standardized yield contract in `contracts/router/src/convert_sy_coin_to_cw20.rs:66-70`, resulting in the principal tokens remaining in the router contract.

Secondly, the `join_dual` function computes the liquidity amount to transfer to the pool in `contracts/router/src/join_dual.rs:54-56`. The issue is that the liquidity amount

required may be less due to the logic in `contracts/notional-pool/src/pool/mod.rs:173-181`, causing the excess to remain in the router contract.

Lastly, the router contract uses approximation parameters (indicated as `ApproxParams` struct) to estimate the amount to send. Due to rounding issues or potential incorrect user inputs, dust amounts may remain (e.g., `ApproxParams.eps` value).

Consequently, the funds left in the router contract can be stolen by others, causing users to lose funds.

Recommendation

We recommend implementing a refund mechanism after every successful operation.

Status: Acknowledged

For the third point raised above, the client states that if approximations are inefficient, they will consider doing refunds. Approximations happen off-chain to guarantee less gas usage.

12. `ListMarkets` query fails if any market's principal token is expired

Severity: Minor

In `contracts/router/src/query.rs:56`, the `ListMarkets` query iterates through all the markets and computes the exchange rate. If any market's principal token expires, an overflow error will occur in `contracts/notional-pool/src/pool/mod.rs:520`, causing the query to fail.

Recommendation

We recommend computing the exchange rate only if the principal token has not expired.

Status: Resolved

13. Incorrect contract label configured for the yield token

Severity: Informational

In `contracts/router/src/admin.rs:122`, the `after_sy_created` function sets the yield token contract label as "sy". This is incorrect because the label is for the standardized yield contract, not the yield token contract.

Recommendation

We recommend modifying the label to “yt”.

Status: Resolved

14. Incorrect `SwapExactSyForYtResponse.net_yt_out` field due to rounding issue

Severity: Informational

In `contracts/router/src/swap_sy_yt.rs:129`, the yield token output amount is computed directly from the simulation output (indicated as `sim_swap_resp.net_yt_out`) and encoded as the `SwapExactSyForYtResponse.net_yt_out` field in `contracts/router/src/swap_sy_yt.rs:180`.

However, the output value may differ from the recipient's actual received amount due to rounding issues, causing the response to show an incorrect output value.

Recommendation

We recommend computing the yield token output amount by the difference between the recipient’s balance before and after the swap.

Status: Acknowledged

Appendix

1. **Test case for "Missing division by zero validation allows attackers to break the standardized yield contract"**

Please run the test case in `contracts/yield-token/tests/send.rs`.

```
#[test]
fn test_dos_sy_contract() {
	use cw_multi_test::{BankSudo, SudoMsg};

	let mut app = App::new(YEAR_DURATION, Decimal::zero());

	let attacker = Addr::unchecked("attacker");

	let token = Coin::new(1_u128, "randomtoken");

	app.app
	.sudo(SudoMsg::Bank(BankSudo::Mint {
		to_address: attacker.to_string(),
		amount: vec![token.clone()],
	}))
	.unwrap();

	app.app
	.send_tokens(attacker.clone(), app.sy_address.clone(), &[token.clone()])
	.unwrap();

	let alice = Addr::unchecked("alice");

	let amount_in = Uint128::new(1_000_000);
	let coin_in = Coin::new(amount_in.u128(), &app.sy_denom);

	app.mint_sy(&alice, amount_in);

	app.execute_yt::<MintResponse>(
	&alice,
	Mint {
		yt_receiver: None,
		pt_receiver: None,
		min_out: Default::default(),
	},
	&vec![coin_in],
	)
	.expect("unexpected error");
}
```

2. **Test case for "Inflated standardized yield token balance causes incorrect reward distribution"**

Please run the test case in `contracts/yield-token/tests/mint_from_cw20.rs`.

```
#[test]
fn test_stuck_rewards() {
	use cw_multi_test::{BankSudo, Executor, SudoMsg};

	let mut app = App::new(YEAR_DURATION, Decimal::zero());

	let alice = Addr::unchecked("alice");
	let alice_deposit_amount = 1_000_000;

	app.mint_sy_cw20(&alice, alice_deposit_amount);

	let yt_addr = app.yt_address.clone();
	let sy_addr = app.sy_address.to_string();
	let resp = app
	.execute::<MintResponse>(
		&Addr::unchecked(sy_addr),
		&alice,
		Send {
		contract: yt_addr.to_string(),
		amount: Uint128::new(alice_deposit_amount),
		msg: to_json_binary(&ReceiveMsg::Mint {
			yt_receiver: None,
			pt_receiver: None,
			min_out: Default::default(),
		})
		.unwrap(),
		}
		.encoded(),
		&vec![],
	)
	.expect("unexpected error");

	assert_eq!(app.yt_balance(&alice), resp.py_out);

	assert_eq!(app.pt_balance(&alice), resp.py_out);

	assert_eq!(app.yt_supply(), resp.py_out);

	assert_eq!(Uint128::zero(), app.asset_balance(&alice));

	// add rewards to sy contract, which will be sent to yt contract
	let reward_amount = 100_000;
	let reward_token = Coin::new(reward_amount, "reward");

	app.app
```

```
.sudo(SudoMsg::Bank(BankSudo::Mint {
      to_address: alice.to_string(),
      amount: vec![reward_token.clone()],
}))
.unwrap();

app.app
.send_tokens(
      alice.clone(),
      app.sy_address.clone(),
      &[reward_token.clone()],
)
.unwrap();

// since alice owns all sy tokens, she should get all the rewards
let sy_total_supply = app
.app
.wrap()
.query_wasm_smart::<balances::TotalSupplyResponse>(
      &app.sy_address,
      &sy::SyQueryMsg::TotalSupply {}.encoded(),
)
.unwrap()
.total_supply;

assert_eq!(app.yt_balance(&alice), sy_total_supply);
assert_eq!(app.pt_balance(&alice), sy_total_supply);
assert_eq!(app.sy_balance(&app.yt_address), sy_total_supply);

// after some time, bob deposits sy tokens
app.app.update_block(|b| {
b.time = b.time.plus_days(2);
});

let bob = Addr::unchecked("bob");
let bob_deposit_amount = alice_deposit_amount;
app.mint_sy_cw20(&bob, bob_deposit_amount);

let sy_addr = app.sy_address.to_string();
app.execute::<MintResponse>(
&Addr::unchecked(sy_addr),
&bob,
Send {
      contract: yt_addr.to_string(),
      amount: Uint128::new(bob_deposit_amount),
      msg: to_json_binary(&ReceiveMsg::Mint {
      yt_receiver: None,
      pt_receiver: None,
      min_out: Default::default(),
      })
```

```
                .unwrap(),
        }
        .encoded(),
        &vec![],
        )
        .expect("unexpected error");

        // however, the rewards will be accrued incorrectly because it includes
bob's stake amount, which is incorrect
        // ultimately alice will get a portion of the rewards, with the remaining
stuck in the contract

        let alice_withdraw_res = app
        .execute_yt::<yield_token::WithdrawResponse>(
                &alice,
                ExecuteMsg::Withdraw {
                receiver: None,
                withdraw_interest: false,
                withdraw_rewards: true,
                },
                &vec![],
        )
        .unwrap();

        let total_shares = alice_deposit_amount + bob_deposit_amount;
        let alice_reward = alice_deposit_amount * reward_amount / total_shares;
        assert_ne!(alice_reward, reward_amount);
        assert_eq!(
        alice_withdraw_res.rewards_withdrawn,
        vec![Coin::new(alice_reward, reward_token.denom)]
        );

        let bob_withdraw_res = app
        .execute_yt::<yield_token::WithdrawResponse>(
                &bob,
                ExecuteMsg::Withdraw {
                receiver: None,
                withdraw_interest: false,
                withdraw_rewards: true,
                },
                &vec![],
        )
        .unwrap();
        assert_eq!(bob_withdraw_res.rewards_withdrawn, vec![]);
}
```