OtterSec

Sandglass

Security Assessment

January 10th, 2025 — Prepared by OtterSec

Ajay Shankar Kunapareddy d1r3wolf@osec.io

Akash Gurugunti sud0u53r.ak@osec.io

Robert Chen r@osec.io

Table of Contents

01 — Executive Summary

Overview

Sandglass Labs engaged OtterSec to assess the `sandglass`, `marginfi-tokenzier`, and `jlp-oracle` programs. This assessment was conducted between December 2nd, 2024 and January 10th, 2025. For more information on our auditing methodology, refer to Appendix B.

Key Findings

We produced 32 findings throughout this audit engagement.

In particular, we identified a critical vulnerability in the instruction responsible for token synchronization between the market's vault and Kamino's vault, allowing unauthorized users to transfer funds from the vault's SY token account (OS-SGL-ADV-00), and a high-risk issue concerning the inability of the Sandglass protocol's freeze functionality to fully restrict critical actions such as trades, swaps, and mints during market freezes (OS-SGL-ADV-02).

Separately, the year and market epoch calculations are unreliable due to their dependence on inaccurate time difference values when the start time does not align with the actual start of the epoch (OS-SGL-ADV-06). Also, the time difference calculation will fail if executed before the market starts, resulting in the reversion of certain pre-market instructions (OS-SGL-ADV-11).

Additionally, the instructions for initializing the Kamino obligation lack validation for the Kamino lending market, allowing unauthorized invocation and potential denial of service by setting mismatched markets. We further highlighted several inconsistencies in the swap withdrawal functionality. These inconsistencies allow trades and redemptions outside the market's active period, enabling unintended swaps before the market starts and unauthorized redemptions before market initiation (OS-SGL-ADV-10), and the failure to subtract the output amounts from the pool amounts, resulting in inaccurate pool state calculations (OS-SGL-ADV-21).

Furthermore, the instruction responsible for updating the market configurations enables admins to modify both static and dynamically updated parameters, creating potential risks of abuse (OS-SGL-ADV-20). Additionally, the utilization of the Solana timestamp leads to errors in fee time calculations when calculating the fee adjustments after the market has ended (OS-SGL-ADV-09).

We also made recommendations for modifying the codebase to improve efficiency (OS-SGL-SUG-06) and suggested streamlining the overall code to reduce complexity and enhance readability (OS-SGL-SUG-04). Moreover, we advised removing unutilized and redundant code within the system to further improve readability (OS-SGL-SUG-08) and incorporating additional safety checks to enhance robustness and security (OS-SGL-SUG-07, OS-SGL-SUG-03).

02 — Scope

The source code was delivered to us in a Git repository at https://github.com/Sandglass-labs/sandglass. This audit was performed against commits `8fe52b8`, `e1c4382`, and `23ce219`.

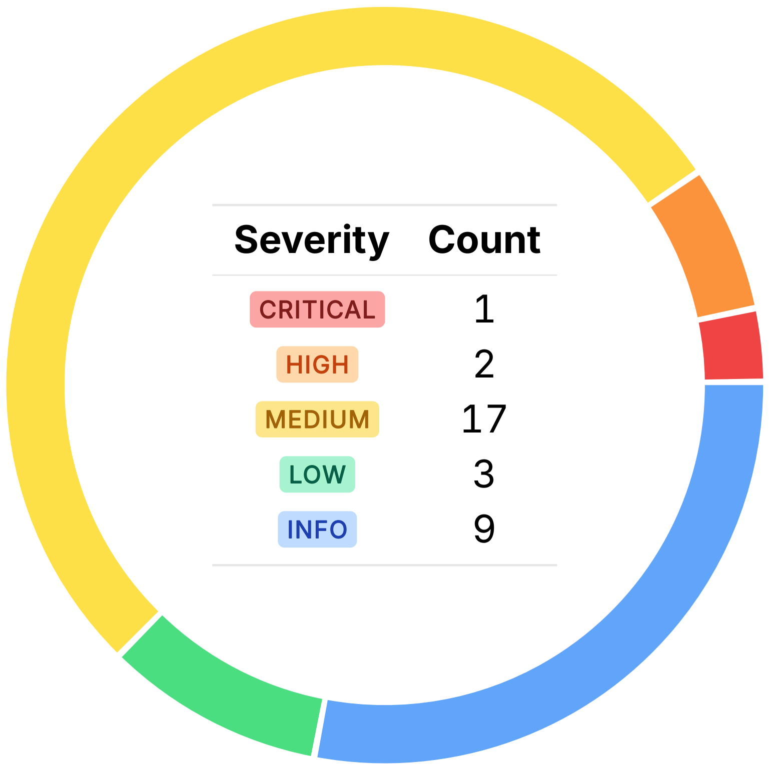
A brief description of the programs is as follows:

Name	Description
sandglass	Sandglass is a pool-based yield trading protocol on Solana. It splits yield-bearing tokens into their principal and yield components, enabling each to be traded separately.
marginfi-tokenizer	It contains instructions for interacting with the MarginFi tokenizer program.
jlp-oracle	It provides accurate and up-to-date price information for JLP (Jupiter Liquidity Pool) assets.

03 — Findings

Overall, we reported 32 findings.

We split the findings into **vulnerabilities** and **general findings**. Vulnerabilities have an immediate impact and should be remediated as soon as possible. General findings do not have an immediate impact but will aid in mitigating future vulnerabilities.



Severity	Count
CRITICAL	1
HIGH	2
MEDIUM	17
LOW	3
INFO	9

04 — Vulnerabilities

Here, we present a technical analysis of the vulnerabilities we identified during our audit. These vulnerabilities have *immediate* security implications, and we recommend remediation as soon as possible.

Rating criteria can be found in Appendix A.

ID	Severity	Status	Description
OS-SGL-ADV-00	CRITICAL	RESOLVED ✓	The `sync_kamino_vault` instruction allows unauthorized users to transfer funds from `vault_sy_token_account`.
OS-SGL-ADV-01	HIGH	RESOLVED ✓	`initialize_kamino_obligation` instruction allows exploitation by sending lamports to prevent proper `user_meta_data` initialization.
OS-SGL-ADV-02	HIGH	RESOLVED ✓	The Sandglass protocol's freeze functionality fails to fully restrict critical actions, such as trades, swaps, and mints, when specific actions on market are frozen.
OS-SGL-ADV-03	MEDIUM	RESOLVED ✓	The `start_epoch` is not set in the `InitializeMarket` instruction, resulting in incorrect `epoch_count` calculations, which affect the accuracy of `market_apy` and `market_end_price` values.
OS-SGL-ADV-04	MEDIUM	RESOLVED ✓	`UpdateOracle` sets `increased_fee_usd` to `pool_info.1` when it is less than `oracle.latest_fee_usd`, which might be incorrect in some cases.
OS-SGL-ADV-05	MEDIUM	RESOLVED ✓	In `UpdateOracle`, it is not checked that `jlp_account` matches the `oracle.jupiter_account`.

OS-SGL-ADV-06	MEDIUM	RESOLVED ✓	The calculation of `year_epoch` and `market_epoch` in `get_market_apy` is unreliable due to its reliance on an inaccurate `time_diff` value when `start_time` does not align with the actual start time of the `start_epoch`.
OS-SGL-ADV-07	MEDIUM	RESOLVED ✓	In `constants`, a dummy mint is utilized for `INF_MINT`, even in non-dummy cases.
OS-SGL-ADV-08	MEDIUM	RESOLVED ✓	The `GetCtokenPrice` instruction fails to call the `ending_pool_accrue_bank_interest` CPI, resulting in inaccuracies in the price calculation.
OS-SGL-ADV-09	MEDIUM	RESOLVED ✓	Utilizing `solana_timestamp` (which defaults to zero if `clock_needed` is false) results in errors in fee time calculations when calculating the fee adjustments in `get_market_data` after the market has ended.
OS-SGL-ADV-10	MEDIUM	RESOLVED ✓	`withdraw_swap` allows trades and redemptions outside the market's active period, enabling unintended swaps and redemptions before the market starts.
OS-SGL-ADV-11	MEDIUM	RESOLVED ✓	In `get_market_apy`, the `time_diff` calculation will fail if it is called before the market starts, resulting in pre-market instructions that utilize `get_market_data` to revert.
OS-SGL-ADV-12	MEDIUM	RESOLVED ✓	The swap functionality in both the `deposit_pool` and `deposit_sy` instructions allows trades to occur before the market starts.

OS-SGL-ADV-13	MEDIUM	RESOLVED ✓	After the market closure, the redemption process returns `SY` based on the last SOL price, rendering residual `SY` and yield fees in the vault without collection mechanisms, and it does not handle cases where the current SOL price is higher than the market's end price.
OS-SGL-ADV-14	MEDIUM	RESOLVED ✓	The `deposit` instruction utilizes outdated pool token amounts and `LP` supply to calculate the pool price, resulting in inaccurate `LP` pricing.
OS-SGL-ADV-15	MEDIUM	RESOLVED ✓	`get_dex_price` lacks checks for oracle staleness, potentially allowing the utilization of stale price data.
OS-SGL-ADV-16	MEDIUM	RESOLVED ✓	In `get_market_apy`, if `market_type != 0`, the `market_apy` remains zero, preventing updates to the market configuration.
OS-SGL-ADV-17	MEDIUM	RESOLVED ✓	The `InitializeMarket` instruction lacks checks to ensure that `PT`, `YT`, and `LP` mints are newly initialized, allowing an admin to pre-mint tokens and retain freeze authority, which allows potential manipulation or arbitrary market halts.
OS-SGL-ADV-18	MEDIUM	RESOLVED ✓	The `UpdateMarketConfig` instruction allows admins to modify both static and dynamically updated parameters, creating potential risks of abuse, including rug pulls, by enabling unauthorized manipulation of critical market variables.
OS-SGL-ADV-19	MEDIUM	RESOLVED ✓	Utilizing pre-initialized `ctoken_mint` and `usdc_escrow` creates a rug pull risk, as their authorities may be maliciously altered after initialization. This, combined with the admin's ability to modify critical configuration parameters, increases the potential for exploitation.

OS-SGL-ADV-20	LOW	RESOLVED ✓	The leftover `PT` and `YT` tokens remain in the pool after the redemption amount is deducted in `withdraw_swap` and after depositing, instead of transferring all tokens to user vaults.
OS-SGL-ADV-21	LOW	RESOLVED ✓	`withdraw_swap` fails to subtract `output_pt_amount` and `output_yt_amount` from the pool amounts when calculating `pool_amount`, resulting in inaccurate pool state calculations.
OS-SGL-ADV-22	LOW	RESOLVED ✓	Liquidity providers may exploit price fluctuations and leftover `PT` / `YT` after swaps, resulting in imbalanced pool proportions and increased gains at the expense of other users.

Unauthorized Fund Drainage CRITICAL OS-SGL-ADV-00

Description

The `sync_kamino_vault` instruction is designed to transfer tokens from `vault_sy_token_account` to `kamino_vault_sy_token_account`. However, it contains a critical security flaw as there is no check to ensure that the caller is an authorized entity. Thus, there are no restrictions on who may call it, enabling anyone to trigger this instruction and potentially drain the funds from `vault_sy_token_account`.

Remediation

Add appropriate access control mechanism to restrict arbitrary users from calling the instruction.

Patch

Fixed by removing the instruction.

Blocking Metadata Initialization HIGH OS-SGL-ADV-01

Description

In the current implementation of `initialize_kamino_obligation`, the initialization of the `user_meta_data` account is skipped if it already holds lamports. This implies that if the `user_meta_data` account has any balance of lamports, the function assumes it is already initialized and skips the `user_meta_data` initialization process. Thus, this may be exploited by sending lamports to the fixed program-derived address, blocking its proper initialization.

>_ *src/instructions/initialize_kamino_obligation.rs* RUST

```rust
pub fn create_obligation(
    &mut self,
    user_lookup_table: Pubkey,
    args: InitObligationArgs,
) -> Result<()> {
    [...]
    if self.user_meta_data.lamports() == 0
        && self.user_meta_data.owner.eq(&self.system_program.key)
    {
        [...]
        let init_user_metadata_data = KaminoInitUserMetadata { user_lookup_table }.pack();
        let init_user_metadata_ix = Instruction {
            accounts,
            program_id: self.kamino_program.key(),
            data: init_user_metadata_data,
        };
        [...]
    }
    [...]
}
```

Furthermore, allowing the initialization of multiple obligations for the market may result in fragmented liquidity, which is both undesirable and unintended.

Remediation

Restrict the access to `initialize_kamino_obligation` to admins only and remove the lamports check from the `user_meta_data` account to avoid the DoS scenario.

Patch

Fixed in 85ae036.

Incomplete Freeze Enforcement Risks HIGH OS-SGL-ADV-02

Description

The current market freeze functionality does not fully restrict actions as intended. Market freezing is a critical risk management feature designed to prevent transactions such as deposits, withdrawals, or swaps during unsafe conditions or protocol upgrades and maintenance. Permitting trades during a freeze may result in market imbalances, invalid financial positions, and unauthorized asset movements. The following actions, which remain allowed under the current freeze logic, should be restricted:

1. The `deposit_pool` and `withdraw_swap` instructions allow trades when trading is frozen.
2. The `deposit_sy` instruction permits users to mint or swap tokens, even when minting or trading is frozen or prior to the market's start.
3. The `swap_sy_pt` and `swap_sy_yt` instructions enable `SY` token minting, allowing users to mint `SY` tokens even when market minting is frozen.
4. The `swap_pt_sy` and `swap_yt_sy` instructions allow the execution of the redeem functionality, enabling users to convert `PT` or `YT` into `SY` tokens even when market redemption is frozen.

Remediation

Update the freeze logic to effectively restrict the aforementioned actions, ensuring the protocol's safety and integrity.

Patch

Fixed in a0d49bb.

Uninitialized Start Epoch MEDIUM OS-SGL-ADV-03

Description

The `start_epoch` field is not set in the `InitializeMarket` instruction, which results in incorrect calculations of the `epoch_count` and subsequently affects the `market_apy` and `market_end_price` . The `start_epoch` field in the `market_account` denotes the starting epoch for the market. This is crucial for calculating the number of epochs that have passed since the market began.

```rust
>_ src/instructions/initialize_market.rs                                RUST

#[derive(AnchorSerialize, AnchorDeserialize)]
pub struct InitMarketArgs {
    start_price: u64,
    market_apy: u64,
    start_time: u64,
    end_time: u64,
    price_base: u64,
    compounding_period: u64,
    max_price_threshold: u64,
}
```

Specifically, in `get_market_apy`, the `epoch_count` depends on the `start_epoch` to determine how many epochs have passed since the market was initiated. `get_market_apy` computes the market's annual percentage yield (APY) and the market end price. Thus, if `start_epoch` is not properly set during market initialization, the calculation of `epoch_count` will be incorrect, resulting in faulty calculations for the APY and the expected market end price.

Remediation

Ensure that the `start_epoch` is properly initialized in the `InitializeMarket` instruction.

Patch

Fixed in 1fba9ea.

Inconsistency in Fee Accumulation Logic MEDIUM OS-SGL-ADV-04

Description

The vulnerability in the `UpdateOracle` instruction in `jlp-oracle` program arises from the logic that calculates the `increased_fee_usd`. The current code sets `increased_fee_usd` to `pool_info.1` if `pool_info.1` is less than `oracle.latest_fee_usd`, assuming this case would arise only if the fee is reset to 0 and starts accumulating again. However, there could be case where the `pool_info.1` is reset to 0 and subsequently increases beyond the value of `oracle.latest_fee_usd` before this instruction is called, which would lead to incorrect accumulated fee calcuation and incorrect price calculations.

>_ *jlp-oracle/src/instructions/update_oracle.rs* RUST

```rust
pub fn handler<'a, 'b, 'c, 'info>(
    ctx: Context<'a, 'b, 'c, 'info, UpdateOracle<'info>>,
) -> Result<()> {
    [...]
    let increased_fee_usd = if pool_info.1 < oracle.latest_fee_usd {
        if DEV_MODE {
            msg!("fee usd reinitialized");
        }
        pool_info.1
    } else {
        if DEV_MODE {
            msg!("fee usd still increasing");
        }
        pool_info.1.checked_sub(oracle.latest_fee_usd).unwrap()
    };
    [...]
}
```

Remediation

Modify `UpdateOracle` to handle the case where the `pool_info.1` is reset to 0 properly.

Patch

Acknowledged by the developers.

Missing Account Validation Check MEDIUM OS-SGL-ADV-05

Description

There is a lack of validation between the `jlp_account` and the `oracle.jupiter_account` in the `UpdateOracle` instruction in the `jlp-oracle` program.

Currently, the `jlp_account` is an unchecked account. There is no validation to ensure that `jlp_account` is the intended account associated with the oracle. The only validation present is the admin signer requirement, which prevents unauthorized access.

Remediation

Ensure that the `jlp_account` passed in the instruction matches the `oracle.jupiter_account` stored in the oracle's state. Also, it is reasonable to make the instruction permissionless once this check is added since all accounts will be properly validated and there are no input parameters.

Patch

Fixed by ensuring `jlp_account` is a fixed address in 6daeecd.

Unreliable Timing Metrics Due to Epoch Misalignment MEDIUM OS-SGL-ADV-06

Description

The vulnerability arises from inaccuracies in the `time_diff` calculation utilized to compute `year_epoch` and `market_epoch` in `get_market_apy`. The `time_diff` is calculated as `epoch_start_timestamp - start_time`. It is intended to represent the duration of time from the start of the epoch (`epoch_start_timestamp`) to the market's `start_time`. If the `start_time` of the market does not align with the beginning of an epoch (`start_epoch`), the `time_diff` calculation becomes unreliable.

>_ src/state/market.rs RUST

```rust
// get the latest market apy and the sol price when the market ends
pub fn get_market_apy(
    market_account: &Market,
    sol_price: u128,
    price_base: u128,
) -> Result<(u128, u128, u128)> {
    [...]
    if solana_timestamp < end_time && market_sol_price < sol_price {
        [...]
        if compounding_period == 0 {
            [...]
            if solana_timestamp > epoch_start_timestamp.checked_add(update_skip_time).unwrap()
                && now_epoch >= last_update_epoch
            {
                epoch_count =
                    (now_epoch as f64) - (market_account.market_config.start_epoch as f64);
                let time_diff = (epoch_start_timestamp as f64) - (start_time as f64);
                year_epoch = (year_time / time_diff) * epoch_count;
                market_epoch = (epoch_count * (market_time as f64)) / time_diff;
            }
        }[...]
    }
    [...]
}
```

If `start_time` is somewhere in the middle of the epoch, `time_diff` only captures the partial time left in that epoch rather than the full epoch length. If `time_diff` is inaccurate, the conversion of `epoch_count` to meaningful time intervals becomes flawed, affecting both `year_epoch` and `market_epoch`.

Remediation

Ensure that `start_time` is initialized to coincide with the `start_epoch`'s starting timestamp.

Patch

Fixed in 1fba9ea.

Utilization of Dummy Mint Address in Production MEDIUM OS-SGL-ADV-07

Description

The issue concerns the handling of the `INF_MINT` constant utilized in `market_price`. This constant is defined in `constants`, which utilize `#[cfg(feature = "dummy")]` and `#[cfg(not(feature = "dummy"))]` conditional compilation attributes to direct the compiler on how to handle the code based on whether the feature is enabled or not.

>_ *src/utl/constants.rs* RUST

```rust
[...]
#[cfg(feature = "dummy")]
pub const INF_MINT: &str = "EuuTuJfGN1roGgkMUjUyyFjBMup2hhXaHD9DN6T68mTJ";
#[cfg(not(feature = "dummy"))]
pub const INF_MINT: &str = "EuuTuJfGN1roGgkMUjUyyFjBMup2hhXaHD9DN6T68mTJ";
[...]
```

Currently, `INF_MINT` is the same for both the dummy (testing) and real-world (production) configurations. This implies that even when the feature `"dummy"` is not enabled, the code will still utilize the dummy mint address rather than the correct, valid one for the production environment.

Remediation

Ensure to utilize the correct address for `INF_MINT` in the production environment.

Patch

Fixed in 8ee133a.

Missing CPI Call for Interest Accrual Calculation MEDIUM OS-SGL-ADV-08

Description

In the `GetCtokenPrice` instruction of the Marginfi tokenizer program, the price of `cTokens` is derived from the total assets in the bank and the `cToken` supply. However, the absence of the `lending_pool_accrue_bank_interest` CPI (Cross-Program Invocation) call before the price calculation creates a significant vulnerability. As interest is periodically accrued on the assets, it affects the total asset value of the lending pool. If the protocol is not consistently accounting for accrued interest, the total asset value may become outdated or inaccurate.

>_ *src/instructions/user/get_ctoken_price.rs* RUST

```rust
impl<'info> GetCtokenPrice<'info> {
    pub fn process(&self) -> Result<f64> {

        let total_asset = self.state.get_total_asset(&self.bank)?;

        let ctoken_supply_i80f48 = I80F48::from_num(self.state.ctoken_supply);
        let ctoken_price = total_asset
            .checked_div(ctoken_supply_i80f48)
            .ok_or(SandglassError::MathOverflow)?;

        Ok(ctoken_price.to_num::<f64>())
    }
}
```

Remediation

Ensure that the `lending_pool_accrue_bank_interest` CPI call is made before calculating the `cToken` price.

Patch

Fixed in e1c4382.

Improper Current Timestamp Utilization MEDIUM OS-SGL-ADV-09

Description

The vulnerability is related to how the `now_time` is calculated and how it may revert if the market has ended and the `fee_needed` flag is set to true. In `get_market_data`, if `fee_needed` is set to true, `now_time` is calculated using the formula: `end_time.checked_sub(solana_timestamp as u64).unwrap()`.

```rust
>_ src/utl/common.rs                                                    RUST

pub fn get_market_data<'info>(
    market_account: &Market,
    oracle_account: &UncheckedAccount<'info>,
    remaining_accounts: &[AccountInfo<'info>],
    clock_needed: bool,
    fee_needed: bool,
) -> MarketData {
    [...]
    if fee_needed {
        let start_time = market_account.market_config.start_time;
        let end_time = market_account.market_config.end_time;
        let market_time = end_time.checked_sub(start_time).unwrap();
        let now_time = end_time.checked_sub(solana_timestamp as u64).unwrap();
        [...]
    }
    [...]
}
```

Consequently, if the market has already ended (in the `WithdrawSwap` instruction), then `solana_timestamp` will be less than the `end_time` (as `solana_timestamp` will be zero if `clock_needed == false`), making `now_time` negative. As a result, `unwrap()` will panic because `checked_sub` will fail when trying to subtract a smaller value from a larger one.

Remediation

Utilize `Clock.unix_timestamp` as the current timestamp instead of `solana_timestamp`, especially when `clock_needed == false`, and handle the underflow case.

Patch

Fixed in ede8ed9 by disallowing any instructions that require market data after the market ends.

Timing Vulnerabilities in Withdrawals MEDIUM OS-SGL-ADV-10

Description

The `withdraw_swap` instruction allows operations that undermine market integrity due to insufficient constraints on market timing. The instruction permits swaps by calculating asset outputs without validating the market state to check if the market has officially started or has already ended. This is problematic because the pricing mechanisms for the tokens depend on market parameters.

Thus, this issue will result in unintended trades. Furthermore, the `withdraw_swap` instruction does not restrict the `redeem` operation to active markets only. Redemptions should be disallowed before the market has begun.

Remediation

Add a check within `withdraw_swap` to ensure that the instruction operates only during the valid market period. Additionally, block redemptions unless the market has officially begun.

Patch

Fixed in ede8ed9.

Failure in Pre-Market Calculations MEDIUM OS-SGL-ADV-11

Description

The vulnerability arises due to how the `epoch_start_timestamp - start_time` operation in `get_market_apy` is utilized without accounting for situations where the market has not yet started. In `get_market_data`, when `get_market_apy` is invoked, it calculates the elapsed epochs and times.

>_ *src/state/market.rs* RUST

```rust
pub fn get_market_apy(
    market_account: &Market,
    sol_price: u128,
    price_base: u128,
) -> Result<(u128, u128, u128)> {
    [...]
    if solana_timestamp < end_time && market_sol_price < sol_price {
        [...]
        if compounding_period == 0 {
            [...]
            if solana_timestamp > epoch_start_timestamp.checked_add(update_skip_time).unwrap()
                && now_epoch >= last_update_epoch
            {
                epoch_count =
                    (now_epoch as f64) - (market_account.market_config.start_epoch as f64);
                let time_diff = (epoch_start_timestamp as f64) - (start_time as f64);
                year_epoch = (year_time / time_diff) * epoch_count;
                market_epoch = (epoch_count * (market_time as f64)) / time_diff;
            }
        }[...]
    }
    Ok((market_apy, market_sol_price, market_end_price))
}
```

However, the `epoch_start_timestamp - start_time` calculation will fail if it is called before the market starts, thereby preventing the execution of `get_market_data`. Thus, pre-market instructions that utilize `get_market_data` will revert.

Remediation

Handle the case where `epoch_start_timestamp - start_time` is performed and an underflow is possible.

Patch

Fixed in 1fba9ea.

Premature Swaps MEDIUM OS-SGL-ADV-12

Description

There is an inconsistency in the `deposit_pool` and `deposit_sy` instructions related to swap operations. Currently, the swap functionality in both the `deposit_pool` and `deposit_sy` instructions allows trades to occur before the market officially starts, which may result in unintended trades and market inconsistencies.

>_ *src/utl/deposit.rs* RUST

```rust
pub fn execute_deposit<'info>([...]){
    [...]
    validate_market_time(&market_account.market_config, false, true)?;
    [...]
}
```

Remediation

Add logic to prevent any swaps before the market's proper activation.

Patch

Fixed in 2dd14a3.

Lack of SY Price Update Post Market End MEDIUM OS-SGL-ADV-13

Description

After the market ends, the `redeem` instruction calculates the `SY` amount based on the last `sol_price`, which is fetched from market data. Any residual `SY` tokens that may remain in the vault do not have any clear reclaiming mechanism. Additionally, there is no method to retrieve the accumulated `yield_fee`, which would normally be derived from `YT`-generated yield. Furthermore, if the market ends and the `sol_price` is higher than the `market_sol_price`, the redemption calculation does not address this properly.

Remediation

Improve the redemption logic after market closure to handle residual tokens and ensure that the protocol properly accounts for cases that may arise after the market ends.

Patch

Acknowledged by the developers.

Missing Value Update MEDIUM OS-SGL-ADV-14

Description

`execute_deposit` in the `deposit` instruction does not utilize the latest pool token amounts and `LP` supply to calculate `pool_price`, resulting in imprecise `LP` price calculations.

>_ *src/utl/deposit.rs* RUST

```rust
pub fn execute_deposit<'info>([...]){
    [...]
    // update lp staking information for the sandglass account
    let pool_price = get_pool_price(
        pool_pt_token_account.amount,
        pool_yt_token_account.amount,
        pt_price,
        yt_price,
        price_base,
        token_lp_mint_address.supply,
    )?;
    [...]
}
```

Remediation

Update the calculation of `pool_price` to leverage the latest pool token amounts and `LP` supply, ensuring accurate `LP` pricing.

Patch

Fixed in 1d76910.

Absence of Staleness Check MEDIUM OS-SGL-ADV-15

Description

In `market_price`, there is no check for oracle staleness within `get_dex_price`. This omission may allow the utilization of outdated or stale price data. The function assumes that the oracle data is current without verifying the time elapsed since the last update.

Remediation

Compare the oracle's last price update time against a maximum staleness threshold stored in the `market`.

Patch

Fixed by implementing appropriate staleness checks for each oracle type.

Discrepancy in Market APY Calculation MEDIUM OS-SGL-ADV-16

Description

Within `market::get_market_apy`, when `market_type != 0`, the `market_apy` value remains zero, preventing updates via `save_market_config`. `market_apy` drives how the market's yield calculations and future prices are updated. If `market_apy` remains zero, it implies that no meaningful APY updates occur, causing the market's yield information to become stale.

Remediation

Ensure that the calculation of `market_apy` works correctly for all `market_type` values.

Patch

Fixed in 858f634.

Insufficient Mint Initialization Checks MEDIUM OS-SGL-ADV-17

Description

There is a lack of validation for the `PT`, `YT`, and `LP` mint accounts and their associated token accounts in the `InitializeMarket` instruction. The code does not verify whether the `PT`, `YT`, and `LP` mints are newly created or pre-existing. Consequently, an admin can utilize existing mints with non-zero supplies, allowing them to pre-mint tokens, transfer authority to the `market_signer`, and initialize the market while retaining freeze authority. This enables the admin to arbitrarily halt market operations or manipulate the market as desired.

Furthermore, `InitializeMarket` does not verify whether the `SY` mint is one of the supported yield token mints during market initialization. When an unsupported `SY` mint is utilized, `get_dex_price` will eventually throw an error while attempting to retrieve the price of the unsupported token. This error will cause any subsequent instructions that rely on the price of the `SY` mint to fail.

Remediation

Initialize the token and mint accounts in the instructions, or verify the following:

1. Mint supplies are set to 0.
2. Decimals match the `SY` mint decimals.
3. Mint and token account authorities are set to `market_signer`.
4. Token account balances are set to 0.
5. Freeze authority is set to None.

   Addtionally, include a verification step where the provided `SY` mint is checked against supported yield token mints.

Patch

Fixed in 37efedd.

Unrestricted Admin Control Risks MEDIUM OS-SGL-ADV-18

Description

The `update_market_config` instruction, in its current implementation, grants the admin unrestricted access to modify all market configuration parameters. This includes parameters that should remain static, such as `price_base`, `start_price`, `start_time`, and `end_time`. Modifying these static values will invalidate earlier assumptions about the market's behavior. Furthermore, the instruction also permits the modification of dynamic variables that are updated automatically by the protocol, such as `market_apy` and `market_end_price`.

```rust
pub fn update_market_config(
    [...]
) -> Result<()> {
    let market_account = &mut self.market_account;
    market_account.market_config.price_base = price_base;
    market_account.market_config.start_time = start_time;
    market_account.market_config.start_price = start_price;
    market_account.market_config.start_epoch = start_epoch;
    market_account.market_config.start_lp_value = start_lp_value;
    market_account.market_config.end_time = end_time;
    market_account.market_config.market_end_price = market_end_price;
    market_account.market_config.market_apy = market_apy;
    [...]
    Ok(())
}
```

Arbitrary changes to these parameters may disrupt market calculations, create unpredictable behavior, or break integrations relying on fixed assumptions. While flexibility is important for adapting to changing market conditions or for handling unexpected events, this degree of access introduces significant centralization risks and potential vulnerabilities, where the admin may execute a rug pull.

Remediation

Limit the `update_market_config` instruction to only modify operationally necessary parameters, and restrict static values from any updates.

Patch

Acknowledged by the developers.

Rug-pull Risk MEDIUM OS-SGL-ADV-19

Description

There is a potential rug-pull risk in the `Initialize` instruction within `marginfi-tokenizer`, specifically due to the utilization of pre-initialized `ctoken_mint` and `usdc_escrow`. The mint ownership may be transferred after token minting or by assigning freeze authority. Similarly, unintended control may be obtained through the `close_authority` on token accounts.

>_ *src/instructions/admin/config.rs* RUST

```rust
impl<'info> Config<'info> {
    pub fn process(&mut self) -> Result<()> {
        [...]
        state.ctoken_mint = *self.mint.to_account_info().key;
        state.marginfi_account = *self.marginfi_account.to_account_info().key;
        state.marginfi_group = *self.marginfi_group.to_account_info().key;
        state.marginfi_bank = *self.marginfi_bank.to_account_info().key;
        state.marginfi_bank_liquidity_vault =
            *self.marginfi_bank_liquidity_vault.to_account_info().key;
        Ok(())
    }
}
```

Similarly, the current implementation of the `Config` instruction (shown above) introduces a rug-pull risk by allowing the admin to modify critical variables, such as `ctoken_mint` and `marginfi` accounts.

Remediation

Utilize program-controlled logic instead of pre-initializing critical accounts, and also implement a multi-signature mechanism to avoid a single entity controlling several critical parameters.

Patch

Fixed in e1c4382 and f546f2c.

Failure to Transfer Residual Token Amount LOW OS-SGL-ADV-20

Description

The issue involves the leftover `PT` / `YT` tokens in the pool after deducting the `redeem_amount`. The expected output of `PT` and `YT` is calculated based on the user's withdrawal request. The `redeem_amount` is determined as the minimum of the expected `PT` and `YT`, effectively allowing the user to redeem these tokens proportionally from the pool. However, after the user redeems the `redeem_amount`, the remaining `PT`/ `YT` tokens remain in the pool instead of being transferred to the user's vaults.

Similarly, during deposits, leftover `PT`/ `YT` is unnecessarily taken from users, benefiting liquidity providers at the expense of users. Additionally, any excess `PT` / `YT` from swapped amounts is added to the pool instead of being returned to the user.

Remediation

Ensure that the leftover amounts of `PT` and `YT` after redeeming the `redeem_amount` and after depositing are transferred to the user's vault, as done in the `withdraw_pool` instruction.

Patch

Acknowledged by the developers.

Inaccurate Pool Amount Calculation LOW OS-SGL-ADV-21

Description

There is an omission in the calculation of the `pool_amount` (via `get_pool_amount`) within the `withdraw_swap` instruction. `get_pool_amount` is responsible for calculating the total pool amount based on the token balances in the respective liquidity pools. When a swap is performed via `withdraw_swap`, certain amounts of `PT` (Principal Tokens) and `YT` (Yield Tokens) are removed from the pools. These amounts are represented by `output_pt_amount` and `output_yt_amount`, respectively.

>_ src/instructions/withdraw_swap.rs RUST

```rust
pub fn withdraw_swap(
    &mut self,
    args: WithdrawSwapArgs,
    remaining_accounts: &[AccountInfo<'info>],
) -> Result<()> {
    [...]
    let pool_pt_amount = u128::try_from(self.pool_pt_token_account.amount).unwrap();
    let pool_yt_amount = u128::try_from(self.pool_yt_token_account.amount).unwrap();

    [...]
    let pool_amount = get_pool_amount(
        self.pool_pt_token_account.amount,
        self.pool_yt_token_account.amount,
        pt_price,
        yt_price,
        price_base,
    )?;
    [...]
}
```

In the current implementation, `get_pool_amount` is called without subtracting the `output_pt_amount` and `output_yt_amount` from their respective pool balances (`pool_pt_token_account.amount` and `pool_yt_token_account.amount`). As a result, it utilizes outdated pool values.

Remediation

Ensure that `output_pt_amount` and `output_yt_amount` are subtracted from their respective pool balances before calling `get_pool_amount`.

Patch

Fixed in ac42f60.

Mismanagement of Leftover Tokens LOW OS-SGL-ADV-22

Description

In the `swap_pt_sy` and `swap_yt_sy` instructions, any leftover `PT` and `YT` tokens after subtracting the `redeem_amount_in` value are merged into the liquidity pool instead of being returned to the user. This benefits liquidity providers when excess tokens are injected into the pool.

>_ *src/instructions/swap_pt_sy.rs* RUST

```rust
pub fn swap_pt_sy(
    [...]
) -> Result<()> {
    [...]
    let swap_amount_pt_in = u128::try_from(amount_in)
        .unwrap()
        .checked_sub(redeem_amount_in)
        .unwrap();
    [...]
    token::transfer(
        CpiContext::new(
            self.token_program.to_account_info(),
            Transfer {
                from: self.user_source.to_account_info(),
                to: self.pool_pt_token_account.to_account_info(),
                authority: self.payer.to_account_info(),
            },
        ),
        u64::try_from(swap_amount_pt_in).unwrap(),
    )?;
    [...]
}
```

Liquidity providers may exploit this behavior by front-running the `swap_pt_sy` and `swap_yt_sy` instructions with swaps to intentionally create imbalances in the `PT` and `YT` proportions, encouraging under-swapping or over-swapping. This further increases their gains while disrupting the pool's token balance and negatively impacting the overall proportions.

Remediation

Ensure any leftover `PT` and `YT` tokens are returned to users.

Patch

Acknowledged by the developers.

05 — General Findings

Here, we present a discussion of general findings during our audit. While these findings do not present an immediate security impact, they represent anti-patterns and may result in security issues in the future.

ID	Description
OS-SGL-SUG-00	The overall security and robustness of admin account management logic may be enhanced by avoiding hard-coded addresses and implementing a two-step process for admin changes.
OS-SGL-SUG-01	There is a risk of denial-of-service if a specific fee token account is compromised or deleted.
OS-SGL-SUG-02	It is essential to verify that the signer is an admin for the `InitKaminoObligation` instruction, as it is a critical one-time operation per market that involves selecting a lending market.
OS-SGL-SUG-03	There are several instances where proper validation is not performed, resulting in potential security issues.
OS-SGL-SUG-04	The overall code may be streamlined further to reduce complexity and enhance readability.
OS-SGL-SUG-05	Suggestions to optimize the code for enhanced efficiency.
OS-SGL-SUG-06	Recommendation for modifying the codebase to improve functionality, efficiency, and maintainability.
OS-SGL-SUG-07	Additional safety checks may be incorporated within the codebase to make it more robust and secure.
OS-SGL-SUG-08	The codebase contains multiple cases of unnecessary code that should be removed for better maintainability and clarity.

Admin Management Enhancement OS-SGL-SUG-00

Description

Utilizing constant addresses to store admin keys reduces flexibility, especially in environments where admin keys may need to change periodically. Updating them in such cases becomes more difficult. Furthermore, directly assigning a new admin in a single step may result in irreversible mistakes if the wrong address is supplied inadvertently, creating a situation where the admin functionality is transferred to an unintended key, resulting in loss of access.

Remediation

Store the admin address directly on the `Market` account instead of relying on predefined constant addresses in the program. Additionally, implement a two-step admin change process where, in the first step, a request proposing a new admin address is submitted. The proposed admin must then explicitly confirm the request to finalize the change, ensuring that the new admin address is correct and authorized.

DOS Risks Due to User Controlled Fee Account OS-SGL-SUG-01

Description

Currently, the `fee_lp_token_account` is utilized for holding fees, which is tied to the user's token account. However, this introduces a risk if the account holding the fees is deleted, compromised, or otherwise becomes inaccessible. Particularly, it may result in a denial-of-service scenario.

>_ *src/instructions/swap_pt_yt.rs* RUST

```rust
#[derive(Accounts)]
pub struct SwapPtYt<'info> {
    [...]
    #[account(
        mut,
        address = market_account.fee_lp_token_account @ ErrorCode::IncorrectAccount,
    )]
    pub fee_lp_token_account: Box<Account<'info, TokenAccount>>,
    [...]
}
```

Remediation

Utilize a protocol-controlled account for `fee_lp_token_account` to ensure the protocol has full control over the account. This approach guarantees that even if a user account is compromised or deleted, the protocol may continue its operations.

Lack of Signer Verification OS-SGL-SUG-02

Description

The `InitKaminoObligation` instruction is critical for initializing a user's obligation in the Kamino lending market. Since it is a one-time operation per market, it associates the market with a specific lending platform. Allowing unauthorized users to execute this function may result in market misconfigurations, as the signer is not currently verified as an admin.

Remediation

Verify that the signer is an admin in the `InitKaminoObligation` instruction.

Missing Validation Logic OS-SGL-SUG-03

Description

1. Add validation for `cur_time < start_time < end_time` in the `InitializeMarket` instruction to verify that the market is initialized to start in the future and that the end time is after the start time.

>_ src/instructions/initialize_market.rs RUST

```rust
pub fn validate(ctx: &Context<InitializeMarket>) -> Result<()> {
    // check the payer for the market creator in production environment
    if !is_update_payer(&ctx.accounts.payer.to_account_info()) {
        return Err(ErrorCode::InvalidPayer.into());
    }

    Ok(())
}
```

2. Currently, the `Freeze` structure defines the fields as `u8`, allowing them to hold any value within the range of an unsigned 8-bit integer. However, the fields are logically binary. Thus, allowing values other than 0 or 1 will result in ambiguity and will be against the design of the protocol. Utilize a boolean type for fields in the `Freeze` structure or validate `update_freeze` inputs to be 0 or 1.

>_ src/state/market.rs RUST

```rust
#[derive(AnchorSerialize, AnchorDeserialize, Clone, Copy, Default)]
pub struct Freeze {
    pub market: u8,
    pub mint: u8,
    pub redeem: u8,
    pub trade: u8,
    pub deposit: u8,
    pub withdraw: u8,
}
```

3. Ensure that the fee numerators are less than the `fee_denominator` in `UpdateFees::validate`. If any numerator exceeds the denominator, the corresponding fee would be greater than 100%, which is undesirable. Hence, restricting numerators to be less than the denominator enforces consistency across all fee configurations. Similarly, add a check for `fee_time_exp <= config_denominator` in `UpdatePoolConfig::validate`.

4. Validate token authorities in all instructions to ensure correct inputs and enable early reversion for errors.

5. In jlp-oracle, verify the owner and discriminator of the `jlp_account` before deserializing it in `utils::get_jlp_pool_info` to ensure the correct account is passed. Alternatively, it may be done while initializing the `JlpOracle` account.

>_ *programs/jlp-oracle/src/util.rs* RUST

```rust
pub fn get_jlp_pool_info(jlp_pool: &UncheckedAccount) -> Result<(u128, u64)> {
    let jlp_pool_data_slice: &[u8] = &jlp_pool.try_borrow_data()?;
    let mut jlp_pool: Box<JLPPool> = Box::new(JLPPool::default());
    {
        let pool_ref = &mut &jlp_pool_data_slice[8..];
        *jlp_pool = JLPPool::deserialize(pool_ref)?;
    }

    Ok((jlp_pool.aum_usd, jlp_pool.pool_apr.realized_fee_usd))
}
```

Remediation

Add the missing validations mentioned above.

Code Clarity

OS-SGL-SUG-04

Description

1. In `State::on_ctoken_mint` and `State::on_ctoken_burn`, utilize checked mathematical operations to improve reliability by throwing custom errors in the event of an overflow or underflow.

   >_ *src/state.rs* RUST

   ```rust
   impl State {
       [...]
       pub fn on_ctoken_mint(&mut self, amount: u64) {
           self.ctoken_supply += amount
       }

       pub fn on_ctoken_burn(&mut self, amount: u64) {
           self.ctoken_supply -= amount;
       }
       [...]
   }
   ```

2. Modify both `swap_calc_pt_in` and `swap_calc_yt_in` to return `None` on errors instead of a `CurveResult` with `destination_amount == 0`. This change simplifies error handling by avoiding the creation of invalid `CurveResult` objects and signaling failure with `None`.

   >_ *src/curve/swap_calc_pt_in.rs* RUST

   ```rust
   pub fn swap_calc_pt_in(
       [...]
   ) -> Option<CurveResult> {
       [...]
       if destination_amount >= pool_yt_amount {
           return Some(CurveResult {
               source_amount,
               destination_amount: 0,
               trade_fee,
               platform_fee,
               after_virtual_pt: virtual_pt,
               after_virtual_yt: virtual_yt,
               is_error: true,
           });
       }
       [...]
   }
   ```

3. Move `increase_stake_pt`, `decrease_stake_pt`, `increase_stake_yt`, `decrease_stake_yt`, `increase_stake_lp`, and `decrease_stake_lp` to the `SandglassAccount` implementation and refactor the `Market` structure accordingly. This will allow the `Market` structure to focus on tracking the overall token pool, while the logic for modifying individual stakes is handled by the `SandglassAccount`.
4. Utilize optional structures for the `clock` and `fee` variables in `get_market_data` rather than defaulting them to zero to improve code clarity and robustness. This approach makes it explicit whether these fields were requested and populated or left intentionally absent. If these fields are used without checking, the default values (`0`) may propagate through calculations, resulting in logical errors or incorrect results.
5. Consolidate all `update_*` admin instruction contexts into a single unified context, encapsulating the admin check instead of defining multiple contexts for each instruction, to reduce code redundancy.

Remediation

Update the code with the above modifications to ensure improved clarity.

Code Optimization OS-SGL-SUG-05

Description

1. Utilize the built-in `std::cmp::min` functionality for the `redeem_amount_in` calculation in the `swap_pt_sy` and `swap_yt_sy` instructions instead of an `if-else` check to make the calculation clearer.

```rust
>_ src/instructions/swap_pt_sy.rs                                        RUST

pub fn swap_pt_sy([...]) -> Result<()> {
    [...]
    if result.destination_amount > redeem_pt_in {
        redeem_amount_in = redeem_pt_in;
    } else {
        redeem_amount_in = result.destination_amount;
    }
    [...]
}
```

2. In `deposit`, `input_yt_amount` and `input_pt_amount` are calculated via standard integer division with `checked_div`. This remainder is calculated separately, and if greater than 0, 1 is added to the result, which gives same result as ceiling division. Therefore, it is recommended to utilize ceiling division directly to efficiently round up the value.
3. `market_price::get_msol_price` may be simplified by utilizing the bitwise operation `>> 32` on `msol_price_hex * price_base` instead of calculating and dividing by `msol_base`, thereby optimizing the price calculation.

```rust
>_ src/utl/market_price.rs                                               RUST

pub fn get_msol_price(oracle_account: &UncheckedAccount, price_base: u128) -> Result<u128>
    ↪ {
    [...]
    let msol_price = msol_price_hex
        .checked_mul(price_base)
        .unwrap()
        .checked_div(msol_base)
        .unwrap();
    [...]
}
```

4. To simplify account structure design, unutilized fields may be replaced with a single padding slice, such as `_padding: [u64; 50]`, to reserve space for future updates. This approach allows the reserved space to be repurposed for new fields during upgrades, streamlining the structure's evolution.
5. Create a custom macro, `dev_msg`, for debug printing in `DEV_MODE`. This will optimize the code by replacing repetitive conditional checks.

Remediation

Integrate the above-listed optimizations into the codebase.

Code Refactoring OS-SGL-SUG-06

Description

1. `initialize_market` is responsible for minting initial token amounts to the pool accounts. If the minted fee LP tokens are redeemable, there is a chance that the liquidity pool may become empty. And to avoid that scenario, it is recommended to send the initial `LP` `mint_amount` to an unreachable address whose owner is trusted not to redeem them or to an address whose owner cannot sign transactions (such as the zero address).
2. To streamline account validation, utilize Anchor's built-in `#[has_one]` attribute to simplify the `address = market_account.*` check, rather than performing a manual address check each time.

   >_ *src/instructions/swap_pt_yt.rs* RUST

   ```rust
   #[derive(Accounts)]
   pub struct SwapPtYt<'info> {
       [...]
       #[account(
           mut,
           address = market_account.fee_lp_token_account @ ErrorCode::IncorrectAccount,
       )]
       pub fee_lp_token_account: Box<Account<'info, TokenAccount>>,
       [...]
   }
   ```

3. Add an admin instruction to modify the immutable `fee_lp_token_account`.
4. The `kamino::deposit_kamino_ctoken` and `kamino::withdraw_kamino_ctoken` implementations independently validate whether the collateral is a supported mint type, creating a risk of inconsistent behavior if only one function is updated when new mints are added. To mitigate this, consolidate the validation into a shared function, ensuring that updates to supported mints are centralized and applied consistently.
5. Store pool balances directly in the `market` account instead of deriving them from token account amounts to eliminate the risk of price manipulation via unauthorized token deposits.
6. Although it is an admin instruction, using a PDA with `jlp_account` as a seed for the oracle account in `InitOracle` within jlp-oracle is recommended to ensure only one oracle exists per jlp account.

Remediation

Incorporate the above-mentioned refactors into the codebase.

Additional Safety Checks OS-SGL-SUG-07

Description

1. In `deposit_sy::validate`, ensure that `mint_amount` is not zero so that invalid mint amounts do not pass through the validation process, thereby preventing unwanted contract states.

   >_ *src/instructions/deposit_sy.rs* RUST

   ```rust
   pub fn validate(ctx: &Context<DepositSY>, args: DepositSYArgs) -> Result<()> {
       // check freeze state
       if ctx.accounts.market_account.freeze.market == 1
           || ctx.accounts.market_account.freeze.deposit == 1
       {
           return Err(ErrorCode::ProgramIsFrozen.into());
       }
       // check input
       if args.lp_amount == 0 || (args.pt_amount == 0 && args.yt_amount == 0) {
           return Err(ErrorCode::InvalidInput.into());
       }
       // check the overall LP mint amount
       if ctx.accounts.token_lp_mint_address.supply == 0 {
           return Err(ErrorCode::InvalidInput.into());
       }
       Ok(())
   }
   ```

2. Compare `args.lp_amount` with `user_lp_token_account` in `withdraw_swap::validate` and `withdraw_pool::validate` to ensure that a user does not attempt to withdraw or swap more `LP` tokens than they actually hold.
3. Modify the `initialize_market` instruction to utilize `get_market_data` for setting `start_price` and `market_apy` instead of accepting user-provided values. This ensures the market is initialized with accurate, real-time data and aids in early error detection.
4. Validate the oracle account owner and discriminator before updating the `oracle_account` field in the `market_account` to ensure it is correctly linked to the `SY` mint associated with the market.

Remediation

Add the checks stated above.

Redundant/Unutilized Code OS-SGL-SUG-08

Description

1. Remove redundant signer accounts in pricing functions and eliminate unutilized or redundant instructions, such as `test_mod`, `redeem_klend_c_token_all`, and `redeem_solend_c_token_all`.
2. In `kamino::deposit_kamino_ctoken`, the `_klend_market_authority_info` parameter is not utilized and should be removed.
3. Remove unutilized fields, such as `start_lp_value`, from `MarketConfig`.
4. In the `State` structure, storing the `marginfi_bank_liquidity_vault` account is unnecessary, as it is a PDA that will be validated during the `marginfi` CPI calls for deposit and withdrawal operations. Additionally, the `usdc_escrow` field is not utilized in the structure.
5. Remove the redundant `u64` to `u64` conversions that are performed using `u64::try_from`.
6. In `utils` within the jlp-oracle, `_multiply_weight`, `_add_remainder`, and `_get_apr` functions seem unutilized and may be removed. Also, the `GetJLPPool` instruction neither updates the state nor returns data, rendering it unnecessary.
7. Remove unutilized functionality for optionally taking `PT` / `YT` from pool token accounts in the `execute_stake_pt` and `execute_stake_yt`.

Remediation

Remove the redundant and unutilized code instances highlighted above.

A — Vulnerability Rating Scale

We rated our findings according to the following scale. Vulnerabilities have immediate security implications. Informational findings may be found in the General Findings.

CRITICAL	Vulnerabilities that immediately result in a loss of user funds with minimal preconditions. Examples: • Misconfigured authority or access control validation. • Improperly designed economic incentives leading to loss of funds.
HIGH	Vulnerabilities that may result in a loss of user funds but are potentially difficult to exploit. Examples: • Loss of funds requiring specific victim interactions. • Exploitation involving high capital requirement with respect to payout.
MEDIUM	Vulnerabilities that may result in denial of service scenarios or degraded usability. Examples: • Computational limit exhaustion through malicious input. • Forced exceptions in the normal user flow.
LOW	Low probability vulnerabilities, which are still exploitable but require extenuating circumstances or undue risk. Examples: • Oracle manipulation with large capital requirements and multiple transactions.
INFO	Best practices to mitigate future security risks. These are classified as general findings. Examples: • Explicit assertion of critical internal invariants. • Improved input validation.

B — Procedure

As part of our standard auditing procedure, we split our analysis into two main sections: design and implementation.

When auditing the design of a program, we aim to ensure that the overall economic architecture is sound in the context of an on-chain program. In other words, there is no way to steal funds or deny service, ignoring any chain-specific quirks. This usually requires a deep understanding of the program's internal interactions, potential game theory implications, and general on-chain execution primitives.

One example of a design vulnerability would be an on-chain oracle that could be manipulated by flash loans or large deposits. Such a design would generally be unsound regardless of which chain the oracle is deployed on.

On the other hand, auditing the program's implementation requires a deep understanding of the chain's execution model. While this varies from chain to chain, some common implementation vulnerabilities include reentrancy, account ownership issues, arithmetic overflows, and rounding bugs.

As a general rule of thumb, implementation vulnerabilities tend to be more "checklist" style. In contrast, design vulnerabilities require a strong understanding of the underlying system and the various interactions: both with the user and cross-program.

As we approach any new target, we strive to comprehensively understand the program first. In our audits, we always approach targets with a team of auditors. This allows us to share thoughts and collaborate, picking up on details that others may have missed.

While sometimes the line between design and implementation can be blurry, we hope this gives some insight into our auditing procedure and thought process.