Kamino Finance

Audit

Presented by:

OtterSec contact@osec.io

Akash Gurugunti sud0u53r.ak@osec.io

Thibault Marboud thibault@osec.io

Robert Chen r@osec.io

Contents

Appendices

01 | Executive Summary

Overview

Kamino Finance engaged OtterSec to perform an assessment of the `kamino-lending` program. This assessment was conducted between August 18th and September 6th, 2023. For more information on our auditing methodology, see Appendix B.

Key Findings

Over the course of this audit engagement, we produced 13 findings in total.

In particular, we have identified several vulnerabilities within the function responsible for refreshing an obligation, including issues with the validation process for the required instruction sequence (OS-KAMI-ADV-01) and the lapse in utilizing the mode parameter, permitting a single farm to undergo multiple refreshes (OS-KAMI-ADV-04).

Furthermore, we highlighted a flawed comparison check, which resulted in the inability to assign a specific elevation group to the lending market (OS-KAMI-ADV-00) and another issue, which resulted in the failure to refresh an obligation when the lending market owner removes the removed the specific elevation group id associated with that obligation (OS-KAMI-ADV-05).

We also recommended implementing a two-step process while updating the owner for a lending market to minimize the risk of inadvertent utilization of the functionality (OS-KAMI-SUG-00). We further suggested the removal of redundant checks and repetitive code blocks to enhance code readability and efficiency (OS-KAMI-SUG-01, OS-KAMI-SUG-03).

02 | Scope

The source code was delivered in a Git repository at github.com/hubbleprotocol/kamino-lending. This audit was performed against commit 88dfca4.

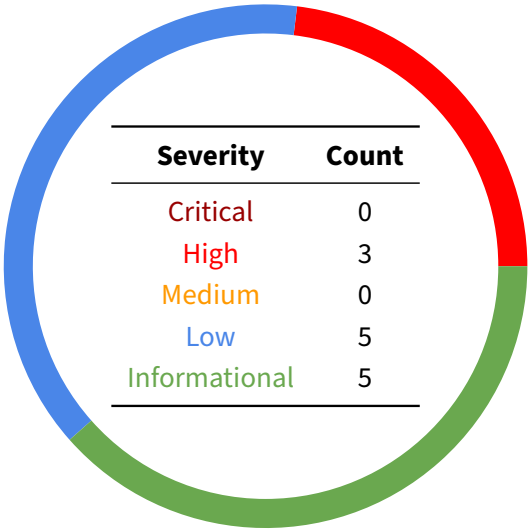
A brief description of the programs is as follows.

Name	Description
kamino-lending	The protocol has been designed to facilitate the borrowing and lending of assets, sharing certain similarities with existing platforms like SPL Lending and Solend protocols. It offers a range of additional features to enhance user experience and the protocol's functionality. These features include specialized farms for managing reserves, where each farm corresponds to a specific kind of collateral or debt, establishing insurance funds for added security, and a mechanism for socializing losses.

03 | Findings

Overall, we reported 13 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	0
High	3
Medium	0
Low	5
Informational	5

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-KAMI-ADV-00	High	Resolved	`refresh_obligation` will fail to execute if the lending market owner removes the specific `elevation_group.id` associated with that obligation.
OS-KAMI-ADV-01	High	Resolved	`check_ixns` utilizes the same mechanism to validate pre and post-instructions, which may not function as intended.
OS-KAMI-ADV-02	High	Resolved	Incorrect process for updating a lending market owner results in the new owner not being assigned as the `farm` admin.
OS-KAMI-ADV-03	Low	Resolved	`handler_update_lending_market` permits the lending market owner to modify elevation group fields without validating them to ensure compliance with the reserve configuration's restrictions.
OS-KAMI-ADV-04	Low	Resolved	`refresh_ix_utils::check_refresh` does not take into account the `mode` parameter, enabling a single `farm` to be refreshed multiple times.
OS-KAMI-ADV-05	Low	Resolved	Inability to assign a specific `elevation_group.id` to the lending market due to a faulty comparison check.
OS-KAMI-ADV-06	Low	Resolved	`withdraw_obligation_collateral` determines the value of `max_withdraw_value`, considering the loan-to-value (LTV) on the withdraw reserve instead of the associated elevation group.
OS-KAMI-ADV-07	Low	Resolved	Absence of a dedicated instruction for facilitating the withdrawal of fees from the `fee_vault`.

OS-KAMI-ADV-00 [high] | Elevation Group ID Mismatch

Description

The vulnerability is related to potential denial of service risks due to a mismatch between the elevation group on an obligation and the available elevation groups on the reserve. This issue arises when an obligation is refreshed before utilizing it or requesting a new elevation group.

lending_operations.rs RUST

```
/* The below check is implemented in both refresh_obligation_deposits and
    ↪  refresh_obligation_borrows with the exception deposit_reserve is changed to
    ↪  borrow_reserve refresh_obligation_borrows.*/

    if elevation_group != ELEVATION_GROUP_NONE
        && !deposit_reserve
            .config
            .elevation_groups
            .contains(&elevation_group)
    {
        return err!(LendingError::InconsistentElevationGroup);
    }
```

In the provided code, when refreshing an obligation, `refresh_obligation` checks if the elevation group specified in the obligation exists in the set of elevation groups for the associated reserve. This check is done within `refresh_obligation_deposits` and `refresh_obligation_borrows`, called internally. If the elevation group specified in the obligation does not exist in the reserve's elevation groups, it results in an error: `(LendingError::InconsistentElevationGroup)`.

The problem arises if the lending market owner, who may modify reserve configurations, removes an elevation group from the reserve associated with existing obligations, which may be done via `update_reserve_config`.

lending_operations.rs RUST

```
pub fn update_reserve_config(
    reserve: &mut Reserve,
    mode: UpdateConfigMode,
    value: [u8; VALUE_BYTE_ARRAY_LEN_RESERVE],) {
    match mode {
     UpdateConfigMode::UpdateElevationGroup => {
            let elevation_group_categories: [u8; 5] =
    ↪  value[..5].try_into().unwrap();
            reserve.config.elevation_groups = elevation_group_categories;
        }
     [...]
    }
}
```

Thus, these obligations become stale in such a scenario because their elevation group no longer exists on the reserve. This may result in a denial of service, as users will be unable to refresh or manage these obligations.

Proof Of Concept

Initially, there is a lending market with multiple elevation groups, including elevation group A and elevation group B. Obligation one specifies elevation group A as its elevation group, and obligation two specifies elevation group B.

1. The lending market owner decides to make changes and removes elevation group B from the reserve configuration via `update_reserve_config`, leaving only elevation group A in the reserve.
2. Now, when obligation two calls `refresh_obligation`, it encounters the following error: `InconsistentElevationGroup` as elevation group B no longer exists on the reserve. As a result, obligation two becomes stale and cannot be managed or refreshed.
3. Obligation one, which specifies elevation group A, is still functional as it matches the remaining elevation group on the reserve.

As a result, users with stale obligations may experience a denial of service. They will be unable to refresh, repay, or liquidate their obligations, effectively locking their funds.

Remediation

Ensure on updating `ElevationGroup`, the previous elevation groups still hold valid for existing obligations associated with them, but with new obligations not being able to select those groups.

Patch

Fixed in PR#110 by adding a flag to disable new loans on an elevation group instead of disabling that elevation group.

OS-KAMI-ADV-01 [high] | Improper Checking Of Instruction Sequence

Description

In `refresh_ix_utils.rs`, `check_ixns` ensures that a specific sequence of instructions, as defined by the `RequiredIx` structures, is executed correctly and meets certain criteria. It ensures that, before and after performing any operation on the reserve concerning a user's obligation, the user's stake is properly refreshed, and all parameters are updated to reflect the latest values, preventing the use of stale values during depositing or borrowing operations on an obligation, or while providing liquidity to reserves.

The issue is related to the discrepancy in the `check_refresh` due to how it checks the required instructions, specifically for `required_post_ixs`. The function uses the same mechanism to validate both pre and post-instructions, which may result in a false validation of the sequence of instructions, causing incorrect updates to obligations and reserves and reducing the financial integrity of the protocol.

refresh_ix_utils.rs RUST

```rust
pub fn check_refresh(
    instruction_sysvar_account_info: &AccountInfo,
    reserves: &[(Pubkey, &Reserve)],
    obligation_address: &Pubkey,
) -> Result<()> {
[...]
    let check_ixns = |required_ixns: Vec<RequiredIx>| -> Result<()> {
        for (i, required_ix) in required_ixns.iter().enumerate() {
            let ix = ix_loader
                .load_instruction_at(
                    current_idx
                        .checked_sub(i + 1)
                        .ok_or(LendingError::IncorrectInstructionInPosition)?,
                )
                .map_err(|_| LendingError::IncorrectInstructionInPosition)?;

            let ix_discriminator: [u8; 8] = ix.data[0..8].try_into().unwrap();
            require_keys_eq!(ix.program_id, crate::id());
            require!(
                ix_discriminator == required_ix.discriminator(),
                LendingError::IncorrectInstructionInPosition
            );
            for (key, index) in required_ix.accounts.iter() {
                require_keys_eq!(
                    ix. accounts
                        .get(*index)
                        .ok_or(LendingError::IncorrectInstructionInPosition)?
                        .pubkey,
                    *key
                );
            }
        }
```

The issue stems from the fact that the function relies on `check_ixns` to validate the sequence of instructions, but `check_ixns` primarily checks the instructions that precede the current instruction index and not the instructions that follow it. Thus, if multiple reserves have different farm configurations, the function may not properly validate the order of instructions across these reserves and may result in a denial of service due to large amounts of instructions that must be processed.

Proof Of Concept

1. Let the current instruction (ix) be a deposit for the obligation collateral.
2. The following is the sequence of instructions that must be executed:
    - not checked `init_obligation_for_farm (s)`.
    - `refresh_reserve (s - multiple reserves in case of liquidation)`.
    - `refresh_obligation`.
    - `refresh_obligation_farms_for_reserve (s)` (collateral && debt) (if has farms).
    - the current instruction.
3. `check_ixns` would validate this sequence of instructions even though no refresh operation has been called post current (ix). I.e., the following instruction was missing: `refresh_obligation_farms_for_reserve (s)`.
4. This may result in an improper update of the obligation parameters where the deposited collateral may not be properly reflected in the obligation.

Remediation

Modify `check_ixns` to consider instructions before and after the current instruction. This ensures the refresh instructions are correctly ordered in the transaction sequence.

Patch

Fixed in PR#115 by adding another struct to determine the type of instructions being checked and checking the instructions accordingly.

OS-KAMI-ADV-02 [high] | Failure To Update Farm Admin

Description

The issue is related to the ownership of a farming pool (farm) being linked to the owner of a lending market and how this linkage is established and updated.

farms_ixs.rs RUST

```rust
pub fn cpi_initialize_farm_delegated(ctx: &Context<InitFarmsForReserve>) ->
    ↪  Result<()> {
    let lending_market = ctx.accounts.lending_market.load()?;
    let lending_market_key = ctx.accounts.lending_market.key();
    let farm_state_key = ctx.accounts.farm_state.to_account_info().key();
    let accounts = farms::accounts::InitializeFarmDelegated {
        farm_admin: ctx.accounts.lending_market_owner.to_account_info().key(),
    [...]
}
```

The `InitFarmsForReserve` instruction sets the farm admin as the lending market owner when initializing the farm-state in `cpi_initialize_farm_delegated` as shown above. This establishes the ownership relationship between the lending market owner and the farm.

klend_client.rs RUST

```rust
pub async fn update_market(
        &self,
        lending_market_pubkey: Pubkey,
        mode: UpdateLendingMarketMode,
        value: UpdateLendingMarketConfigValue,
    ) -> Result<()> {
        let lending_market: LendingMarket = self
            .client
            .get_anchor_account(&lending_market_pubkey)
            .await?;
        let tx = self.client.tx_builder().add_anchor_ix(
            &self.config.program_id,
            kamino_lending::accounts::UpdateLendingMarket {
                lending_market_owner: lending_market.lending_market_owner,
                lending_market: lending_market_pubkey,
            },
            kamino_lending::instruction::UpdateLendingMarket {
                mode: mode as u64,
                value: value.to_bytes(),
            },
        );
        [...]
    }
```

However, this ownership is not automatically updated if the owner of the lending market changes later, as seen in the above function. This may result in a situation where the farm is still controlled by the previous owner of the lending market, even after the ownership of the lending market has been transferred.

This creates a mismatch where the lending market and farm ownership are not synchronized. In a properly functioning system, when you change the owner of the lending market, the ownership of associated farms should also be updated to match the new lending market owner.

Proof Of Concept

1. A lending market is operated by owner A.
2. A farm is initialized and associated with the above lending market via `InitFarmsForReserve`.
3. `cpi_initialize_farm_delegated` which takes `InitFarmsForReserve` as an argument sets the owner of the farm as the owner of the lending market, i.e., farm admin is set to A.
4. Later, ownership of the lending market is transferred from owner A to owner B by executing `update_market`.
5. After the ownership transfer, the lending market now belongs to owner B, but the farm is still administered by owner A because the relationship was established when the farm was initialized.

Remediation

Update the farm admin when the ownership of the lending market changes.

Patch

The Kamino Finance team acknowledged the issue and decided to add a CLI tool to modify the farm admin before updating the lending market owner.

OS-KAMI-ADV-03 [low] | Inconsistent Checks On Elevation Group

Description

The issue relates to potential inconsistencies in the protocol's verification of elevation groups, loan-to-value (LTV) ratios, and liquidation threshold values. Elevation groups categorize assets based on their risk profiles, where each elevation group has specific loan-to-value ratios and liquidation thresholds. Loan-to-value ratios determine how much collateral a borrower must maintain relative to their borrowed amount, while liquidation thresholds represent the point at which a borrower's collateral-to-debt ratio triggers liquidation. Assets or elevation groups with higher risk may have lower liquidation thresholds.

lending_operations.rs RUST

```rust
pub fn validate_reserve_config(config: &ReserveConfig, market: &LendingMarket) ->
    ↪ Result<()> {
        for elevation_group_id in config.elevation_groups {

            let elevation_group = get_elevation_group(elevation_group_id, market)?;
            if elevation_group_id == ELEVATION_GROUP_NONE {
                // The reserve is removed from an elevation group id
            } else {
                [...]
                if elevation_group.liquidation_threshold_pct <
    ↪ config.liquidation_threshold {
                    msg!("Invalid liquidation threshold, elevation id liquidation
    ↪ threshold must be greater than the config's");
                    return err!(LendingError::InvalidConfig);
                }
                if elevation_group.ltv_ratio_pct < config.loan_to_value_ratio {
                    msg!("Invalid ltv ratio, cannot be bigger than the ltv ratio");
                    return err!(LendingError::InvalidConfig);
                }
            }
        }
    }
```

While assigning elevation group IDs to the reserves configuration, the values in the elevation group are checked with the loan-to-value and liquidation threshold values on the reserves configuration, as shown in the code above.

However, the lending market owner has the authority to change the loan-to-value ratios and liquidation thresholds for elevation groups. This introduces a potential inconsistency because, during the modification of these values by the owner, there is no mechanism to ensure that existing reserves and obligations comply with the new values, as seen in the code snippet below. This may result in situations where reserves and borrowers are no longer adequately collateralized or are at risk of liquidation.

handler_update_lending_market.rs RUST

```rust
pub fn process(
    ctx: Context<UpdateLendingMarket>,
    mode: u64,
    value: [u8; VALUE_BYTE_ARRAY_LEN_MARKET],
) -> Result<()> {
UpdateLendingMarketMode::UpdateElevationGroup => {
            let elevation_group: ElevationGroup =
                BorshDeserialize::deserialize(&mut &value[..]).unwrap();
            msg!("Value is {:?}", elevation_group);
            [...]
            if elevation_group.liquidation_threshold_pct >= 100
                || elevation_group.ltv_ratio_pct >= 100
                || elevation_group.ltv_ratio_pct >
    ↪  elevation_group.liquidation_threshold_pct
                || elevation_group.max_liquidation_bonus_bps > FULL_BPS as u16
            {
                return err!(LendingError::InvalidElevationGroupConfig);
            }
            [...]
        }
    }
```

Remediation

Implement a validation mechanism that verifies the validity of the new loan-to-value ratios and liquidation thresholds set by the lending market owner for elevation groups.

Patch

The Kamino Finance team acknowledged the issue and decided to create a tool in the CLI to validate these values.

OS-KAMI-ADV-04 [low] | Unchecked Mode Parameter

Description

The vulnerability in `check_refresh` may allow a user to refresh the same collateral or debt `farm` twice while avoiding the refresh of the other `farm` associated with the reserve.

A reserve may have multiple farms associated with it. Each farm corresponds to a specific kind of collateral or debt. `Farms` manage collateral and debt associated with reserves. `check_refresh` checks the correctness and sequence of instructions related to these farms.

refresh_ix_utils.rs RUST

```rust
pub fn check_refresh(
    instruction_sysvar_account_info: &AccountInfo,
    reserves: &[(Pubkey, &Reserve)],
    obligation_address: &Pubkey,
) -> Result<()> {
[...]
    let check_ixns = |required_ixns: Vec<RequiredIx>| -> Result<()> {
        for (i, required_ix) in required_ixns.iter().enumerate() {
            let ix = ix_loader
                .load_instruction_at(
                    current_idx
                        .checked_sub(i + 1)
                        .ok_or(LendingError::IncorrectInstructionInPosition)?,
                )
                .map_err(|_| LendingError::IncorrectInstructionInPosition)?;

            let ix_discriminator: [u8; 8] = ix.data[0..8].try_into().unwrap();
            require_keys_eq!(ix.program_id, crate::id());
            require!(
                ix_discriminator == required_ix.discriminator(),
                LendingError::IncorrectInstructionInPosition
            );
            for (key, index) in required_ix.accounts.iter() {
                require_keys_eq!(
                    ix.accounts
                        .get(*index)
                        .ok_or(LendingError::IncorrectInstructionInPosition)?
                        .pubkey,
                    *key
                );
            }
        }
        Ok(())
    };
    [...]
}
```

In the provided code, when iterating over farms for a reserve, the `RequiredIx` structure is being constructed, but it does not consider the specific farm type (collateral or debt) or the associated farm state account.

As a result, the code only checks the refresh of a farm based on its position in the loop but not based on the farm type or associated farm state account. This means a user may refresh the same collateral or debt farm multiple times while avoiding the refresh of the other farms associated with the reserve.

Remediation

Add `(reserve.get_farm(farm_type), 4)` to the accounts vector in `required_ix` to ensure all the reserve farms are properly refreshed.

Patch

Fixed in PR#115 by adding `(reserve.get_farm(farm_type), 4)` to the accounts vector in `required_ix`.

OS-KAMI-ADV-05 [low] | Discrepancy In Elevation Group

Description

In `consts.rs`, the constant `MAX_NUM_ELEVATION_GROUPS` is assigned a value of ten, representing the maximum number of elevation groups in the lending market. Each market is associated with certain elevation groups with an ID ranging from one to ten.

handler_update_lending_market.rs RUST

```rust
pub fn process(
    ctx: Context<UpdateLendingMarket>,
    mode: u64,
    value: [u8; VALUE_BYTE_ARRAY_LEN_MARKET],
) -> Result<()> {
UpdateLendingMarketMode::UpdateElevationGroup => {
            let elevation_group: ElevationGroup =
                BorshDeserialize::deserialize(&mut &value[..]).unwrap();
            msg!("Value is {:?}", elevation_group);

            // Check the elevation group id is valid
            if elevation_group.id >= MAX_NUM_ELEVATION_GROUPS {
                return err!(LendingError::InvalidElevationGroupConfig);
            }
            [...]
        }
    }
```

However, within `handler_update_lending_market::process`, when updating the elevation groups, it is observed that the `elevation_group.id` is constrained to a maximum value of nine. This constraint implies that the highest possible value for `elevation_group.id` is nine, consequently limiting the maximum index for storing elevation groups in the `LendingMarket.elevation_groups` to eight. Therefore, based on the mentioned restriction, elevation groups may only accommodate a maximum of nine values, which deviates from the intended limit of ten and restricts the users from using the last elevation group.

Remediation

Modify the check to ensure it fails only if `elevation_group.id` is greater than `MAX_NUM_ELEVATION_GROUPS`.

Patch

Fixed in PR#110 by modifying the above-mentioned check.

OS-KAMI-ADV-06 [low] | Inconsistent Calculation Of Max Withdraw Value

Description

`withdraw_obligation_collateral` utilizes the loan-to-value (LTV) ratio from the `withdraw_reserves` configuration to calculate `max_withdraw_value`, which may not align with the user's obligation if the obligation's elevation group is different from that of the reserve, resulting in incorrect maximum limits and a loss of funds for the user. `obligation.allowed_borrow_value` is calculated based on the loan-to-value on the elevation group; thus, it is best to maintain consistency and utilize the value specified in the associated elevation group.

lending_operations.rs RUST

```rust
pub fn withdraw_obligation_collateral(
    lending_market: &LendingMarket,
    withdraw_reserve: &Reserve,
    obligation: &mut Obligation,
    collateral_amount: u64,
    slot: Slot,
    withdraw_reserve_pk: Pubkey,
) -> Result<u64> {
    [...]
    let max_withdraw_value =
    obligation.max_withdraw_value(Rate::from_percent(loan_to_value_ratio_pct))?;
    [...]
    }
```

Proof Of Concept

Suppose a lending protocol has two elevation groups: group A and group B. User A has an obligation associated with group A, and they want to withdraw collateral from a reserve (call it reserve X). reserve X has its own loan-to-value ratio configured in its settings.

1. `withdraw_obligation_collateral` is called to withdraw collateral from user A's obligation. It utilizes the loan-to-value ratio from reserve X's configuration to calculate `max_withdraw_value`.
2. However, since user A's obligation is associated with group A, the loan-to-value ratio they should follow is the one specific to group A, not the generic loan-to-value ratio of reserve X. The protocol allows different loan-to-value ratios for different elevation groups.
3. If the loan-to-value ratio for group A is different from that of reserve X, user A may be unable to withdraw as much collateral as they should based on their obligation's configuration. They may withdraw less collateral than expected or encounter an error indicating that the withdrawal amount exceeds the maximum allowed by configuration of reserve X.

Remediation

Ensure `withdraw_obligation_collateral` utilizes the loan-to-value ratio specific to the elevation group associated with the user's obligation.

Patch

Fixed in PR#110 by considering the loan-to-value ratio of the elevation group associated with the user's obligation.

OS-KAMI-ADV-07 [low] | Lack Of Withdraw Functionality

Description

To ensure the proper operation of the lending market, include clear instructions for the lending market owner about the withdrawal of the fees that have been collected into `fee_vault` as currently, there exists no mechanism to extract the deposited fee from `fee_vault`. This may result in any deposited fee being permanently locked up in `fee_vault`, making it unusable.

Remediation

Implement instructions on facilitating fee withdrawal.

Patch

Fixed in PR#112 by adding an instruction for the lending market owner to withdraw funds from the fee vault.

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-KAMI-SUG-00	Utilize a two-step verification process to confirm a change in ownership.
OS-KAMI-SUG-01	Removal of redundant checks to improve code readability and efficiency.
OS-KAMI-SUG-02	Time-weighted average price calculation in `pyth` and `scope` is unnecessary.
OS-KAMI-SUG-03	Repetitive code in `switchboard::get_switchboard_price` and `scope::get_price_usd`.
OS-KAMI-SUG-04	An incorrect constant is utilized in `utils/prices/mod.rs`.

OS-KAMI-SUG-00 | Double Verification For Owner Change

Description

In `handler_update_lending_market::process`, the owner of the lending market is updated via `UpdateLendingMarketMode`. However, the owner change is currently a single-step process; there is no confirmation step. Once the transaction is submitted, the owner change is irreversible. This may result in a denial of service when the current owner accidentally sends an unintended input as a parameter while executing an owner change.

handler_update_lending_market.rs RUST

```rust
pub fn process(
    ctx: Context<UpdateLendingMarket>,
    mode: u64,
    value: [u8; VALUE_BYTE_ARRAY_LEN_MARKET],
) -> Result<()> {
    let mode = UpdateLendingMarketMode::try_from(mode)
        .map_err(|_| ProgramError::InvalidInstructionData)?;

    let market = &mut ctx.accounts.lending_market.load_mut()?;

    match mode {
        UpdateLendingMarketMode::UpdateOwner => {
            let value: [u8; 32] = value[0..32].try_into().unwrap();
            let value = Pubkey::from(value);
            market.lending_market_owner = value;
            msg!("Value is {:?}", value);
        }
        [...]
    }
    [...]
}
```

Remediation

Utilize a two-step process to change the owner of the lending market.

Patch

Fixed in PR#111 by utilizing a two-step process to change the owner.

OS-KAMI-SUG-01 | Removal Of Redundant And Unused Code

Description

The following suggestions are regarding the removal of redundant checks and unused code in the `kamino-lending` code base:

1. There are numerous redundant checks within `lending_checks`, as well as redundant PDA validation checks related to lending market authority. These validations are already conducted once by the anchor in the instruction definition and are explicitly executed in `lending_checks`.
2. In `UpdateInsuranceFundDepositor` instruction, given that the `insurance_fund_depositor` account is not utilizing `init_if_needed`, it is unnecessary to execute `load_init` on it and subsequently handle errors to employ `load_mut`. Instead, `load_mut` may be directly employed.
3. In the `SocializeLoss` instruction, the `reserve_destination_liquidity` account passed to it seems unused and may be removed.

Remediation

Ensure to remove the redundant checks within `lending_checks` as well as any redundant PDA validation checks, and in `UpdateInsuranceFundDepositor`, `load_mut` may be directly employed since `insurance_fund_depositor` account is not utilizing `init_if_needed`. Additionally, remove the `reserve_destination_liquidity` account from the `SocializeLoss` instruction.

Patch

Fixed in PR#113 by removing all the redundant checks.

OS-KAMI-SUG-02 | Unnecessary Conditional Calculation

Description

In `pyth` and `scope`, the time-weighted average price calculation persists even when the `token_info.is_twap_enabled()` condition evaluates to false. This behavior appears to be unnecessary and may result in computational overhead. This occurs as currently there is no mechanism implemented in `get_pyth_price_and_twap` and `get_scope_price_and_twap` to evaluate if only the price or both price and the time-weighted average price must be calculated.

pyth.rs & scope.rs RUST

```rust
pub(super) fn get_pyth_price_and_twap(
    pyth_price_info: &AccountInfo,
) -> Result<TimestampedPriceWithTwap> {
    [...]
    let price = price_feed.get_price_unchecked();
    let twap = price_feed.get_ema_price_unchecked();
    [...]
    Ok(TimestampedPriceWithTwap {
        price: price.into(),
        twap: Some(twap.into()),
    })
}

pub(super) fn get_scope_price_and_twap(
    scope_price_account: &AccountInfo,
    conf: &ScopeConfiguration,
) -> Result<TimestampedPriceWithTwap> {
    let scope_prices = get_price_account(scope_price_account)?;
    let price = get_price_usd(&scope_prices, conf.price_chain)?;
    let twap = if conf.has_twap() {
        get_price_usd(&scope_prices, conf.twap_chain)
            .map_err(|e| msg!("No valid twap found for scope price, error: {:?}",
    ↪  e))
            .ok()
    }
    [...]
    Ok(TimestampedPriceWithTwap { price, twap })
}
```

Remediation

Utilize the boolean value obtained from the `token_info.is_twap_enabled()` check as an argument in `get_pyth_price_and_twap` and `get_scope_price_and_twap`, within the `get_most_recent_price_and_twap` function. This approach would allow for conditional time-weighted average price calculation, thereby reducing computational overhead when it is not enabled for the token.

Patch

Fixed in PR#112 by lazy loading the `twap` price only if it is enabled in the config.

OS-KAMI-SUG-03 | Code Repetition

Description

In `switchboard::get_switchboard_price`, the code segment responsible for converting the price to decimal may be substituted with the utilization of the pre-existing `utils::price_to_decimal`, eliminating repetitive code. This same optimization may also be implemented in `scope::get_price_usd`.

switchboard.rs & scope.rs RUST

```rust
fn get_switchboard_price(
    switchboard_feed_info: &AccountInfo
) -> Result<TimestampedPrice> {
        let price_load = Box::new(move || {
            [...]
            Ok(Decimal::from_scaled_val(scaled_value))
    });
}

fn get_price_usd(
    scope_prices: &ScopePrices,
    tokens_chain: ScopeConversionChain,
) -> Result<TimestampedPrice> {
        let price_load = Box::new(move || {
            [...]
            Ok(Decimal::from_scaled_val(scaled_value))
    });
}
```

Remediation

Eliminate the repetitive code blocks utilizing `utils::price_to_decimal` to adhere to best coding practices.

Patch

Fixed in PR#112 by using the same code block.

utils.rs RUST

```rust
/// Lossy conversion from Price to Decimal
///
/// Price is stored with n decimals, Decimal with 18 decimals.
/// If Price has more decimals than Decimal, the extra decimals are lost.
pub fn price_to_decimal(price: Price) -> Decimal {
    // Decimal stored with fixed 18 decimals
```

```
    // Scale price to match 18 decimals.
    let scaled_value = to_scaled_normalized(&price, MAX_DECIMAL_EXPONENT.into());

    Decimal::from_scaled_val(scaled_value)
}
```

OS-KAMI-SUG-04 | Incorrect Usage Of Constant

Description

In `utils/prices/mod.rs`, it is advisable to change the constant named `MIN_CONFIDENCE_PERCENTAGE` to `MAX_CONFIDENCE_PERCENTAGE`, as this constant represents the upper limit for the confidence percentage rather than the minimum value.

mod.rs RUST

```rust
/// validate price confidence - confidence/price ratio should be less than 2%
const MIN_CONFIDENCE_PERCENTAGE: u64 = 2u64;

/// Confidence factor is used to scale the confidence value to a value that can be
    ↪  compared to the price.
const CONFIDENCE_FACTOR: u64 = 100 / MIN_CONFIDENCE_PERCENTAGE;
```

Remediation

Replace `MIN_CONFIDENCE_PERCENTAGE` with `MAX_CONFIDENCE_PERCENTAGE`.

Patch

Fixed in PR#112 by renaming the variable.

A | Vulnerability Rating Scale

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

Critical	Vulnerabilities that immediately lead to 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 could lead to 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 could lead to denial of service scenarios or degraded usability. Examples: • Malicious input that causes computational limit exhaustion • Forced exceptions in normal user flow
Low	Low probability vulnerabilities which could still be exploitable but require extenuating circumstances or undue risk. Examples: • Oracle manipulation with large capital requirements and multiple transactions
Informational	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 implementation of the program 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 sum, 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 get a comprehensive understanding of 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 the other 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.